ARL-TR-8247 ● DEC 2017

US Army Research Laboratory

Generating Atomistic Slab Surfaces with Adsorbates

by Joshua T Paul and Krista R Limmer

Approved for public release; distribution is unlimited.

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-8247 ● DEC 2017

US Army Research Laboratory

Generating Atomistic Slab Surfaces with Adsorbates

by Joshua T Paul
Oak Ridge Institute for Science and Education, Oak Ridge, TN

Krista R Limmer
Weapons and Materials Research Directorate, ARL

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
December 2017	Technical Report	May 2017–August 2017

4. TITLE AND SUBTITLE
Generating Atomistic Slab Surfaces with Adsorbates

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Joshua T Paul and Krista R Limmer

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-WMM-F
Aberdeen Proving Ground, MD 21005-5069

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-8247

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report details the high-throughput slab generation and molecular adsorption toolkit developed during the High Performance Computing Modernization Program's FY17 internship program under project number HIP-17-029. The toolkit was developed to aid in corrosion-resistant magnesium alloy design and uses high-fidelity density functional theory calculations to predict and evaluate the effect of potential secondary phases on the cathodic corrosion reaction thermodynamics. The framework consolidates available open source tools such as genetic algorithms, crystal structure databases, and slab generation tools in conjunction with a newly developed molecular adsorbate placement tool.

15. SUBJECT TERMS
high throughput, density functional theory, magnesium, corrosion, surface reaction, adsorption

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Krista R Limmer
Unclassified	Unclassified	Unclassified	UU	92	19b. TELEPHONE NUMBER (Include area code) 410-306-2039

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures v

Acknowledgments vi

1. Introduction 1

2. Methods of Surface Investigation 2

2.1 Bulk Phase Identification 2

2.2 Slab Generation 3

2.3 Slab Convergence 3

2.4 Adsorption and Binding Energy 4

3. Results and Discussion 5

3.1 Generating Slabs 6

3.1.1 Slab Convergence Testing 6

3.1.2 High-Throughput Slab Generation from Genetic Algorithms 7

3.1.3 High-Throughput Slab Generation from the Materials Project Database 10

3.2 Performing Clean Slab Calculations 10

3.2.1 Selective Dynamics 10

3.2.2 Surface Energy Analysis 11

3.3 Molecule Adsorption on Surfaces 12

3.3.1 Defining Possible Adsorption Sites 12

3.3.2 Placing Adsorbates: Atop Sites 13

3.3.3 Placing Adsorbates: Bridge Sites 14

3.3.4 Placing Adsorbates: Interstitial Sites 15

3.3.5 Generating Adsorption Site List 16

3.4 Adsorption Analysis 16

4. Conclusions 17

5. References 18

Approved for public release; distribution is unlimited.

Appendix A. Software Installation in a Virtual Environment 21

Appendix B. Genetic Algorithm for Structure Prediction (GASP) Input Files 25

Appendix C. Vienna ab initio Simulation Package (VASP) Files 29

Appendix D. Adsorbates.py Software Variable Details 33

Appendix E. Python Script for Complete adsorbates.py Source 45

Appendix F. MPInterface Modification 73

Appendix G. Python Script for reciprocal.py to Generate KPOINT Files 79

List of Symbols, Abbreviations, and Acronyms 83

Distribution List 84

List of Figures

Fig. 1 The phase diagrams generated for Mg-Al by a GA (left) and from the Materials Project database (right). As shown, GAs can be used to identify a much larger number of secondary phases than available in the Materials Project database. 3

Fig. 2 Schematic of *makeGA_slabs*, *makeMP_slabs*, and *makeLigSurface* directory and file organization. Directories generated by *makeGA_slabs* and *makeMP_slabs* are filled solid blue and files are white. Directories generated by *makeLigSurface* are filled solid green. Black dashed lines indicate subdirectories and files generated through these scripts. *makeGA_slabs* and *makeMP_slabs* are run from the parent directory to generate the subsequent files and directories, while *makeLigSurface* is run from inside the POSCAR_n directories. 9

Fig. 3 Side and top-down view of a structure file generated by *oneAtomAdsorb*. The adsorbate (hydrogen atom, brown) adsorbed onto the top of a material (aluminum slab, blue representing aluminum atoms), specifically above a single atom. 14

Fig. 4 Side and top-down view of a structure file generated by *twoAtomAdsorb*. The adsorbate (hydrogen atom, brown) adsorbed onto the top of a material (aluminum slab, blue representing aluminum atoms), specifically between 2 aluminum atoms. 15

Fig. 5 Side and top-down view of a structure file generated by *threeAtomAdsorb*. The adsorbate (hydrogen atom, brown) adsorbed onto the top of a material (aluminum slab, blue representing aluminum atoms), specifically at the center of a triangle formed by 3 aluminum atoms. 16

Acknowledgments

This work was supported in part by JT Paul's appointment as a graduate research participant at the US Army Research Laboratory (ARL) administered by the Oak Ridge Institute for Science and Education through an interagency agreement between the US Department of Energy and ARL.

This work was supported in part by a grant of computer time from the Department of Defense (DOD) High Performance Computing Modernization Program (HPCMP) at the US Air Force Research Laboratory DOD Supercomputing Resource Center. The authors gratefully acknowledge both the grant of computing resources as well as JT Paul's support through the DOD HPCMP internship program.

1. Introduction

Due to its light weight and high strength, magnesium is a desirable material to use in structural design; however, it is highly susceptible to corrosion. Though ultrahigh-purity magnesium does not readily facilitate self-corrosion, chemical defects often act as strong catalysts to the cathodic reaction and promote corrosion.[1] Specifically, the cathodic reaction for magnesium corrosion is reduction of water to OH^- and H^+ balanced by the evolution of H_2 gas. Alloying is currently being examined as a tool to reduce the cathodic reaction kinetics and thereby reduce the overall corrosion rate.[2–4] Because of the low solubility of most alloying additions in magnesium, secondary phases and their influence on the cathodic reaction are of primary interest for corrosion-resistant alloy design.

Design of a novel magnesium alloy based on the propensity for specific surface reactions requires high-throughput, high-fidelity calculations. High-fidelity density functional theory (DFT) calculations allow for accurate parameter-free evaluation of the alloying effect on the critical surface reactions. Investigation of surface reactions in a high-throughput manner requires a complex suite of tools that does not currently exist. Because of the novel alloy chemistry aspect, the stability of secondary phases is empirically unknown and may be addressed through a combination of data mining current materials structure databases and prediction of phases through genetic algorithms (GAs). To investigate the surface reactions on these secondary phases, surfaces must be generated from the bulk crystal structures of interest for an alloy system. Finally, surface reaction thermodynamics may be considered, requiring the placement of reactants and products on the surfaces and sampling multiple configurational spaces to determine the stable configurations.

In this work, several algorithms have been developed to facilitate high-throughput investigations of surface reaction thermodynamics for magnesium alloys. These algorithms are united in a single Python file (*adsorbates.py*). The algorithms include facilitation of slab generation, slab convergence tests, and adsorbate placement on surfaces. The *adsorbates* framework enables a user to evaluate the cathodic reaction thermodynamics on possible secondary phases for a specified alloy system, thereby informing the user of the impact of the secondary phase on cathodic reaction stability. This is accomplished for a user-specified alloy system of interest by 1) importing possible secondary phases from a database or genetic algorithm search, 2) cleaving slabs of various orientations from each bulk structure, 3) generating a grid of possible adsorption sites at and near the slab surface, and 4) calculating the adsorption energy of a molecule as a function of spatial orientation on the possible adsorption sites. This framework was developed to coordinate the passing of structure information throughout the process and it can be easily tailored

Approved for public release; distribution is unlimited.

to other application spaces as well. The details of the algorithms, as well as installation of the necessary tools and initial detail on the operation of GAs to explore magnesium alloys systems is described herein. Appendixes are included to guide users through the software installation process (Appendix A), installation and use of a genetic algorithm (Appendix B), recommended Vienna ab initio Simulation Package (VASP) parameters (Appendix C), and the details of the developed code (Appendix D). The full Python script of *adsorbates* is included as Appendix E, with Appendixes F and G further providing modified versions of 2 existing Python scripts for adsorption schemes k-point grid generation.

2. Methods of Surface Investigation

2.1 Bulk Phase Identification

Stable and metastable compounds that may exist in the alloy of interest have the potential to impact the corrosion-relevant surface reactions. To identify the known and unknown stable and metastable compounds, 2 approaches may be used independently or in combination. The first approach is to use structures from known databases such as the Materials Project[5] that may include experimentally observed compounds as well as computationally derived compounds. The second approach is to explore a composition space for stable and metastable compounds using GAs. In a combined approach, stable and metastable compounds from a database can also be used to "seed" a GA.

The framework developed here has been built to accommodate input from both the Materials Project database as well as searches conducted with the genetic algorithm for structure prediction (GASP)[6]. Additional databases and GAs may also be used with the framework but have not been included at this time. Stable compounds, the compounds used to generate the thermodynamic hull as shown in Fig. 1, should generally be considered as possible phases. Bulk metastable compounds may also be considered and are identified by a negative enthalpy of formation (more stable than its isolated pure components) but a positive hull distance (not as stable as other known compounds in the system). These metastable compounds can appear in experimental samples if processing conditions are modified to favor the formation of this phase. These conditions include creating kinetic restriction (e.g., quenching) or changing the pressure of the synthesis environment. When choosing compounds from which to generate slabs, consideration must be given to whether metastable phases should be included, and if so, how unstable the metastable phases are allowed to be. It is recommended that the metastability not exceed 50 meV/atom for bulk compounds, as higher levels of instability are unlikely to not only be synthesized but also remain stable.[7]

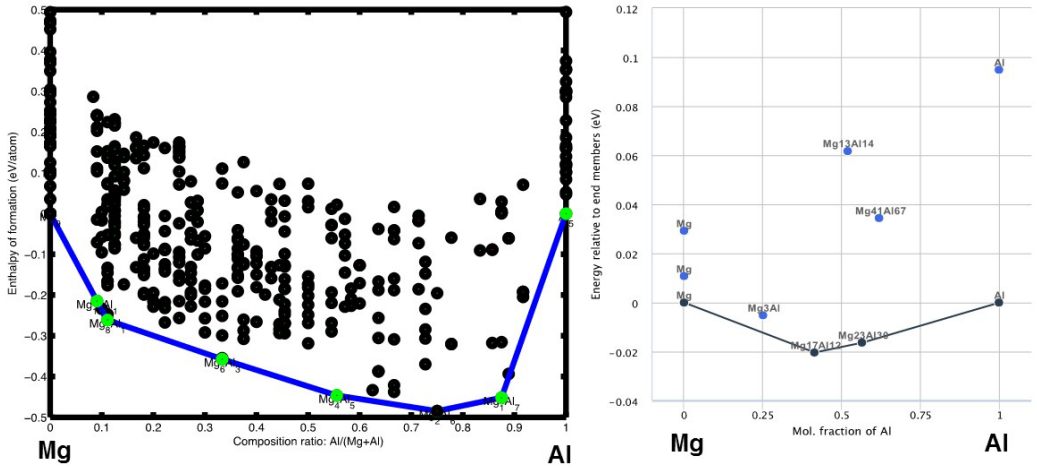

Fig. 1 The phase diagrams generated for Mg-Al by a GA (left) and from the Materials Project database (right). As shown, GAs can be used to identify a much larger number of secondary phases than available in the Materials Project database.

2.2 Slab Generation

Slabs may be generated from any given bulk structure for a range of surface orientations. As described in Section 2.1, 2 approaches exist for identifying bulk compounds from which to generate slabs: using structures from known databases, such as the Materials Project, and using structures identified by a GA. Scripts were created for both approaches and are named *makeMP_slabs* and *makeGA_slabs*, respectively.

Some surface orientations may have more than one unique surface termination. Depending on the surface termination, slabs with varying surface energies can be created. The ability to automatically discern multiple surface termination options for a given orientation is not incorporated at this time. Users are encouraged to use prior knowledge of the systems being investigated when manually adjusting this option.

2.3 Slab Convergence

Prior to investigating adsorbates on surfaces, several steps must occur to prepare the surface. The slab thickness and vacuum spacing must be converged for the simulation model to be relevant. As slab thickness is increased, the surface energy decreases until the inner atoms behave identically to bulk and do not increase the normalized energy of the system. Further increasing the slab thickness does not cause the inner atoms to change in behavior; thus the number of atoms simulated can be minimized, saving computational expense, by identifying the slab thickness

at which this occurs. In addition, some minimum vacuum spacing is required to prevent spurious stabilization. If there is not enough vacuum spacing, the surfaces of the slab interact with each other through the periodic boundary condition, resulting in an incorrect lower system energy. For plane-wave based DFT codes, such as VASP[8] or Quantum ESPRESSO,[9] large vacuum spaces are computationally expensive and should be minimized when possible.

To determine the minimum slab and vacuum thickness, the surface energy is converged. The surface energy is calculated by Eq. 1

$$E_{surface} = (E_{slab} - N_{slab} * E_{bulk})/2\,, \tag{1}$$

where E_{slab} is the energy of the slab, N_{slab} is the number of atoms in the slab, and E_{bulk} is the energy of the bulk material. The division by 2 is a result of there being 2 surfaces on a slab. This value can be normalized by dividing by the surface area of the slab, which is necessary for comparing the energy of different surfaces, and is performed by this algorithm when more than one surface orientation is compared. An additional optional complexity to the creation of surfaces is the option to use selective dynamics to constrain the bottom half of the slab and only optimize the top half of the slab, which is available in this code.

For the purposes of slab convergence, slabs of various thickness and with various vacuum spacing need be calculated. This can occur in serial or simultaneously. If performed in serial, the vacuum spacing should be converged before the slab thickness. If a dipole is present in a material, then the surface energy will diverge. Preliminary results for a fully optimized slab (no selective dynamics) indicate that a slab approximately 19 Å thick (constructed from approximately 10 close-packed layers of atoms) and a vacuum spacing of 15 Å is sufficient for obtaining reasonable energy convergence values of 0.1 meV. Since these thicknesses are system dependent, some systems may give converged results at lower or higher values. In general, the minimum vacuum spacing is more system agnostic.

2.4 Adsorption and Binding Energy

After calculating the surface energies, adsorption simulations can be performed on all of the surfaces generated through *makeMP_slabs* or *makeGA_slabs*, or only those with low surface energy. Adsorbates are placed on the surface of a slab using the *makeLigSurface* function. The adsorption energies of these adsorbates is given in Eq. 2 as

$$E_{adsorption} = (E_{slab+adsorbate} - E_{slab} - E_{adsorbate})\,, \tag{2}$$

where $E_{slab+adsorbate}$ is the energy of the simulation containing the slab and adsorbate, E_{slab} is the energy of the isolated slab, and $E_{adsorbate}$ is the energy of the isolated adsorbate. Higher adsorption energies mean the adsorbate is likely to be bound to the surface of the material.

Adsorption energy can be calculated in 3 different simulation environments: vacuum, implicit solvation, and explicit solvation. Vacuum places the adsorbate on the slab surface while being surrounded by vacuum padding and is the traditional approach. This is the least computationally expensive method, but may not provide accurate binding energies for solvated scenarios. Implicit solvation fills the vacuum spacing with a specific dielectric constant to simulate a given medium. This better captures the adsorbate binding energy, but still does not work well for charged molecules (such as OH^- or H^+). Explicit solvation places water molecules where the vacuum previously existed and may consist of either completely filling the vacuum space or introducing only a single or double layer of solvation around relevant surfaces and molecules. This explicit solvation results in more accurate prediction of behavior but significantly increases computational cost.

Comparison of the adsorption energies of all species involved in a reaction allows for prediction of the reaction through thermodynamics. However, energetic barriers must be calculated using density functional perturbation theory or the nudged elastic band method to determine the kinetics and mechanisms of how a species will migrate across the material.

3. Results and Discussion

The framework developed here to facilitate high-throughput investigations of surface reaction thermodynamics has been compiled into a single python file named *adsorbates.py* that consists of multiple stand-alone algorithms. The logic and function of each of these algorithms is described here. The details of the variables can be found in Appendix D.

Determination and acquisition of the stable and metastable compounds is the first step in this high-throughput process and the incorporation of existing methods into this framework is straightforward. Bulk crystal structures may either be sourced from existing databases such as the Materials Project or generated using a GA such as GASP. Details on the installation and use of GASP to generate crystal structures in accord with this algorithm are provided in Appendix B. Additional details may be found in the GASP manual[10] and usage documentation.

3.1 Generating Slabs

The slab generation scheme is dependent on whether the slab is being generated for initial convergence testing or high-throughput processing. Additionally, separate functions have been developed for high-throughput slab generation for bulk structures sourced from either a database or GA search. Accordingly, 3 different functions have been developed for generating slabs. The first function, *makeConvergeSlabs*, is intended to be used when a new alloy system or application space is being explored to determine the minimum slab dimensional requirements for accurate calculations. Following the convergence determination, slabs meeting the dimensional criteria can be generated for further investigation in a high-throughput manner using *makeGA_slabs* or *makeMP_slabs*, depending on the bulk structure source.

3.1.1 Slab Convergence Testing

The *makeConvergeSlabs* function is designed to automate the process of converging slabs and should be used whenever a new alloy system or application space is being explored. This script should be run on a subset of bulk structures that capture the diversity of the system. The script generates slabs of varying thicknesses and varying vacuum spacings to identify the minimum slab and vacuum thickness to use in the high-throughput slab generation algorithms described in Sections 3.1.2 and 3.1.3.

Slabs are created in set intervals of thickness because they are generated from crystallographic unit cells with defined lattice parameters. For example, if the unit cell height is 5 Å, it is not possible to generate a slab that is 7 Å thick without breaking the crystallographic symmetry. As such, the algorithm identifies what the interval spacing between slab thicknesses will be and generates slabs at these precise intervals, the number of intervals being determined by the user. The first thickness may not be in an interval, but all subsequent thicknesses will be in intervals. Slabs can be generated for any Miller indices (hkl), although the (001) surface is the default.

The vacuum padding does not have the same crystallographic limitations and can be evaluated for the range of values of interest for the user. Although the optimization of the slab thickness and vacuum padding can be done simultaneously, it is more computationally conservative to first converge the vacuum spacing to ensure the surfaces are electronically isolated and then converge the slab thickness. Selective dynamics can be added to these slabs to lower the minimum slab thickness using *addsd*, described in more detail in Section 3.2.1.

Running *makeConvergenceSlabs* will populate the directory it is run from with newly created directories following the nomenclature '[SlabThickness]_[VaccuumPadding]'. Because this nomenclature does not contain any surface orientation or composition information, evaluation of additional surface orientations or bulk structures should be initiated in a different directory. Bulk structure files (POSCAR) do not necessarily need to be contained within the directory from which *makeConvergenceSlabs* is executed and can be identified using a path string input.

After the slabs are generated, they should be structurally optimized within VASP to compare the relaxed surface energies. The submission of these jobs is not automated, as there is not a single way to submit jobs across all job schedulers. However, the VASP "noz" binary should be used during these and all subsequent structural optimization calculations to prevent the simulation cell from resizing in the z-direction and incorrectly impacting the surface energy results. Additional details on the VASP "noz" binary are provided in Appendix C.

The *orgDataConvergence* script was developed to aid in the analysis of the slab convergence simulation results. This script should be run in the same directory where *makeConvergeSlabs* was run. Bulk energy values (eV) are read from the VASP OSZICAR file and are converted to surface energies (eV/Å^2) that are written into a slab convergence results file called "resultsC.txt". This file writes the energetic data generated for slab convergence testing according to increasing slab and vacuum thicknesses. Only systems that successfully terminate on an ionic convergence will appear in the "resultsC.txt" file.

In order to enable accurate surface investigations, the surface energies should be converged to 0.1 meV accuracy. If multiple compounds or surface orientations are used to determine the slab convergence criteria, the largest values for both slab thickness and vacuum spacing should be carried forward as the minimum values for the remainder of the high-throughput investigation of that system.

3.1.2 High-Throughput Slab Generation from Genetic Algorithms

The *makeGA_slabs* function is designed to take materials discovered through GAs and generate slabs of various orientations. Additional details of the installation and recommended application of GASP for this framework are provided in Appendix B. After completing the GA search and identifying the bulk compounds of interest, the *makeGA_slabs* script can be used to generate slabs using the previously determined slab dimensional minimums for a range of surface orientations specified by (hkl) indices. At the time of publishing, the down-selection of compounds from the GASP search for surface investigation is not

automated, requiring the user to manually decide which compounds from the search should be used.

Because the GASP output structure files are all by default named POSCAR and are located in parent directories with non-intuitive names, the structure files for the compounds of interest must be copied to a directory named "sources" and uniquely renamed for clarity. For example, if the compound is the $Mg_{17}Al_{12}$ beta-phase commonly found in the AZ series of magnesium alloys, then the POSCAR could be renamed "Mg17Al12".

Running the *makeGA_slabs* function in the parent directory containing "sources" will result in the creation of a set of directories in the current directory matching the name of each renamed POSCAR located in "sources" as shown in Fig. 2. Inside each of these directories will be subdirectories containing POSCAR files representing slab structures. The name of each subdirectory corresponds to the orientation, in hkl notation, of the slab structure file contained within. For example, a cut along the (111) plane will be labeled "111". Prior to generating the subdirectories and writing the POSCAR files, the symmetry of the slabs is examined using the python materials genomics (pymatgen)[11] *StructureMatcher* tool for duplicates. Duplicate slabs are removed and only the unique slab structure files are written.

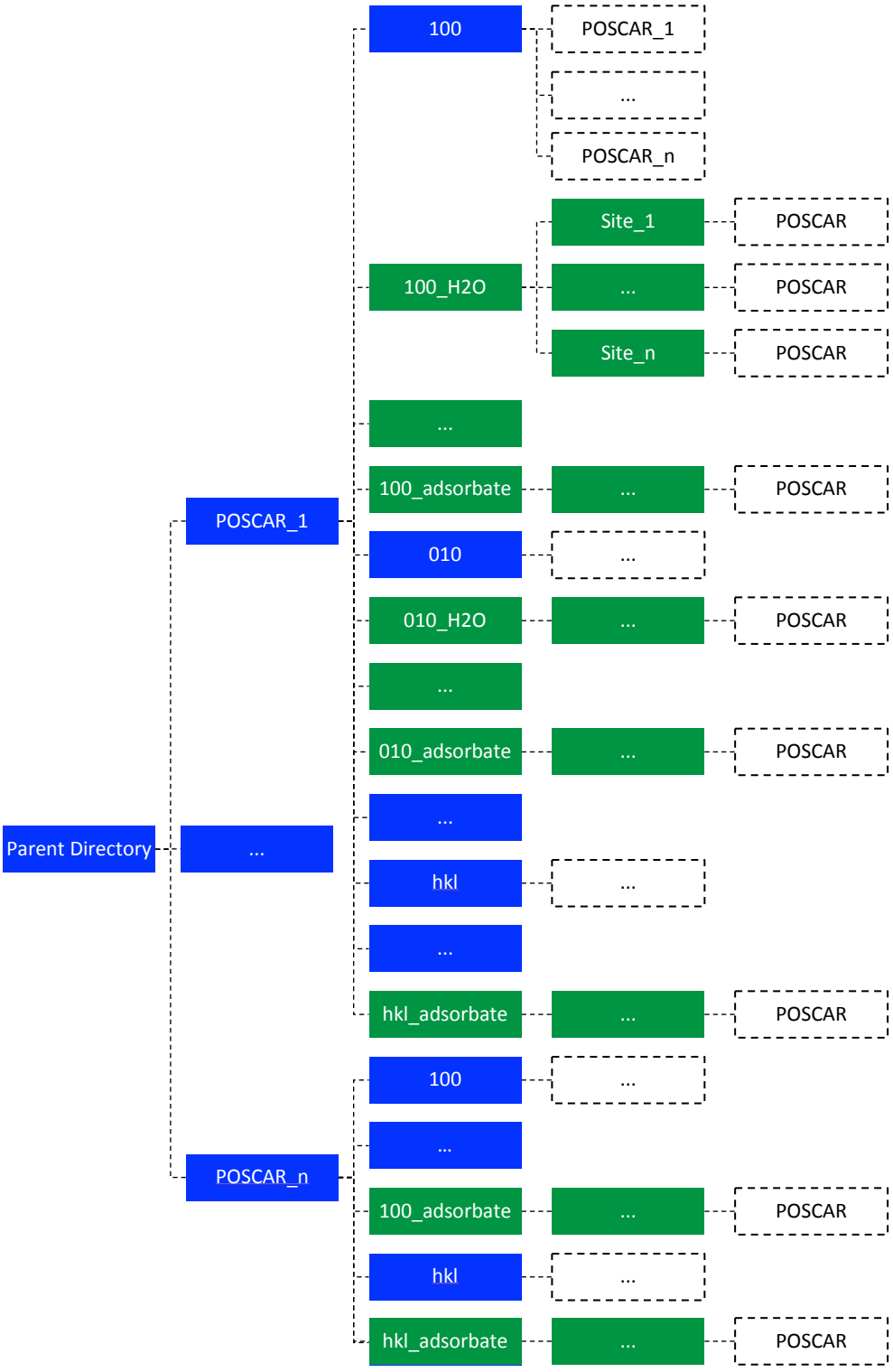

Fig. 2 Schematic of *makeGA_slabs*, *makeMP_slabs*, and *makeLigSurface* directory and file organization. Directories generated by *makeGA_slabs* and *makeMP_slabs* are filled solid blue and files are white. Directories generated by *makeLigSurface* are filled solid green. Black dashed lines indicate subdirectories and files generated through these scripts. *makeGA_slabs* and *makeMP_slabs* are run from the parent directory to generate the subsequent files and directories, while *makeLigSurface* is run from inside the POSCAR_n directories.

The default range of Miller indices for slab surface orientations is between –1 and 1 for each of the h, k, and l indices. However, additional orientations or a larger range of indices may be specified by the user. Some orientations may have potential for multiple surface terminations. Although there is no automated method to search for these additional terminations, expert users can generate these alternate terminations by duplicating the initial bulk POSCAR file and shifting the position of each atom along the direction the slab is being cut.

3.1.3 High-Throughput Slab Generation from the Materials Project Database

The *makeMP_slabs* function uses the Materials Project database to identify bulk compounds for the high-throughput surface investigations. It can take any number of elements as input, and will return all compounds within a defined distance from the thermodynamic hull (including compounds on the hull). When the function is run, the settings can either be entered upon request from the function or entered prior to execution as a list variable. If the list is empty, then the settings are entered as active inputs. Similar to generating slabs from GA searches, the orientation range defaults to a maximum and minimum Miller index of 1 and –1 but can be modified by the user. Additionally, symmetry is evaluated using the pymatgen[11] *StructureMatcher* tool for all of the possible orientations and only unique slabs are written to a directory corresponding to the orientation of the cut.

When run, *makeMP_slabs* will generate several directories named after the bulk compounds selected from the Materials Project database. If 2 compounds share the same name, a "_#" will follow the compound name, with # being a positive integer greater than 1. Each compound directory will contain orientation-named subdirectories that each contains the slab POSCAR structure file of that slab orientation.

3.2 Performing Clean Slab Calculations

Prior to the addition of adsorbates on slab surfaces, the surface energy of the clean slabs should be calculated and the slab surfaces should be structurally optimized. At this point in the framework, a decision should be made regarding the use of selective dynamics.

3.2.1 Selective Dynamics

The function *addsd* adds selective dynamics tags to a VASP structure file. Selective dynamics is the option to restrict the ability of some atoms to move during relaxation. In this function, selective dynamics is used to freeze the atoms in the

bottom half of the slab, preventing motion in the x-, y-, and z-directions. The remaining atoms are all permitted full motion. The halfway point is determined by comparing the lowest and highest z-coordinates in the structure file. If there is an adsorbate on a slab, the middle point will be defined between the bottom of the slab and the top of the adsorbate rather than the bottom and top of the slab. In the event that this results in an undesirable shift in the frozen atom list, the exact point below which the atoms are frozen can be modified by the user. The advantage of selective dynamics is to decrease computation time in 2 ways. First, and most apparent, decreasing the number of atoms able to move results in fewer local optimizations to calculate. Secondly, but a potentially larger impact overall, is that by freezing the bottom half of the slab during the convergence testing and carrying throughout the process, a thinner slab may be used throughout by reaching the energetic convergence criteria sooner due to only optimizing 1 free surface rather than 2.

The *addsd* function may be run at any time in the high-throughput process. It must be run in the directory that contains the POSCAR to be modified to add selective dynamics. Then, the function is simply run with the necessary inputs to add selective dynamics to the POSCAR. If the POSCAR file already has selective dynamics tag present, the halfway point will be recalculated and the tags will be updated.

Caution should be used when reporting surface energies while using selective dynamics to freeze in a bulk structure on the bottom half of a slab. Because the 2 surfaces are no longer equivalent, the division by 2 in Eq. 1 is no longer sufficient to describe the surface energy of the top optimized surface. In addition, Eq. 1 is no longer valid if the surface terminations in the material are not equivalent (i.e., they have differing atomic organizations or elemental compositions).

3.2.2 Surface Energy Analysis

Similar to the *orgDataConvergence* function, the *orgDataSlabs* function collects and organizes data for slabs generated by *makeMP_slabs* or *makeGA_slabs*. This function can be run after a slab generation was begun by *makeGA_slabs* or *makeMP_slabs* and should be run from that same parent directory. Total energy values of each compound and surface orientation considered are read from the OSZICAR file and converted to slab surface energies that are printed in a results file named “resultS.txt”. Any calculations that did not converge prior to termination will not appear in the “resultsS.txt” file. The data are separated based on the material directory and orientation subdirectory to avoid confusion between surfaces of differing materials. Recall that symmetry operations were used to reduce the number of surfaces simulated, so the number of orientations present in

"resultsS.txt" is likely to be less than the maximum possible 27 for a triclinic system cut along the default hkl values ranging from –1 to 1.

3.3 Molecule Adsorption on Surfaces

Adsorbates are placed on slabs using the *makeLigSurface* class, which contains 5 functions that are detailed in Sections 3.31 through 3.3.5. *makeLigSurface* is dependent on a modified version of the MPInterface package.[12] The first function, *setUp*, is run using a slab (or bulk) structure as input to generate the necessary parameters for adding adsorbates. The following 3 functions (*oneAtomAdsorb*, *twoAtomAdsorb*, and *threeAtomAdsorb*) place adsorbates on the slab, returning slabs with an adsorbate placed on the surface. The fifth function, *writeStructs*, runs all of the generated adsorbate-covered slabs through the pymatgen[11] *StructureMatcher* tool, grouping symmetric structures and allowing the user to minimize the number of calculations performed. Two additional helper functions within the class are also defined and used within the other functions. The *getAngles* helper function returns an angle between 2 vectors. The *getAbsCoord* helper function obtains the position of atoms in Cartesian coordinates rather than fractional.

3.3.1 Defining Possible Adsorption Sites

The *setUp* function is designed to set up the future calculations for adsorption placement. It begins by generating a slab from the input structure or creating a structure object and then continues by checking the area of the surface of the slab. If it is less than the desired surface area to prevent adsorbate interaction with its periodic image, the slab is scaled to approximately reach the minimum area.

The next step in the algorithm is to identify all atoms that exist in the top layer of the material. The top layer of the material is defined as any atom that has no other atoms directly above it, where "above" refers to the surface normal.

To account for the possibility that 2 atoms are above each other, but do not appear to be based on the fractional coordinates, transformations of one atom along the *a* and *b* vectors of the unit cell are also considered. If one atom is found to be above the other after a transformation, then the uppermost atom is put under consideration for being a top atom. If an atom was previously under consideration, but another atom was found to be above it, the first atom is removed from consideration while the second is added.

The depth below the topmost layer of surface atoms that may be considered "surface" can also be tailored by the user. This surface depth should be monitored, as the depth at which a hydrogen atom can reach into a surface is not the same as a

water molecule, and prescreening such unreasonable situations can save computation time.

This function returns a list that is used as an input for *oneAtomAdsorb*, *twoAtomAdsorb*, and *threeAtomAdsorb*. In addition, it provides recommended values for the maximum distance into the material that adsorbates can be placed, and the maximum area formed by 3 atoms in *threeAtomAdsorb*. The details of the objects in the list are given in Appendix D. The default distance and area values may not be ideal for all adsorbate sizes. For example, hydrogen atoms can fit into smaller interstitial sites than a water molecule.

The *setUp* function should always be run as a first step in adsorption additions, prior to *oneAtomAdsorb*, *twoAtomAdsorb*, and *threeAtomAdsorb*. It is recommended to use a slab as the input structure rather than generate one using this function so the user is sure of what structure they are adding adsorbates to. Additionally, the final output of this function is the same slab as input or generated but with the previously bottom surface now the top surface. This is useful for situations where the surface terminations of the same orientation are not always equivalent.

3.3.2 Placing Adsorbates: Atop Sites

The *oneAtomAdsorb* function takes the output list of *setUp* with other settings and returns a list of slabs with adsorbates as structure objects as the output. The adsorbate positions considered are only directly above atoms in atop sites as shown in Fig. 3. The adsorbate can also be rotated with respect to the adsorbate site or the atom of the adsorbate meeting the slab surface. This distance from the adsorbate and the surface atom it is above can be specified by the user. The function returns a list of unique structures as determined by the pymatgen[11] *StructureMatcher* tool.

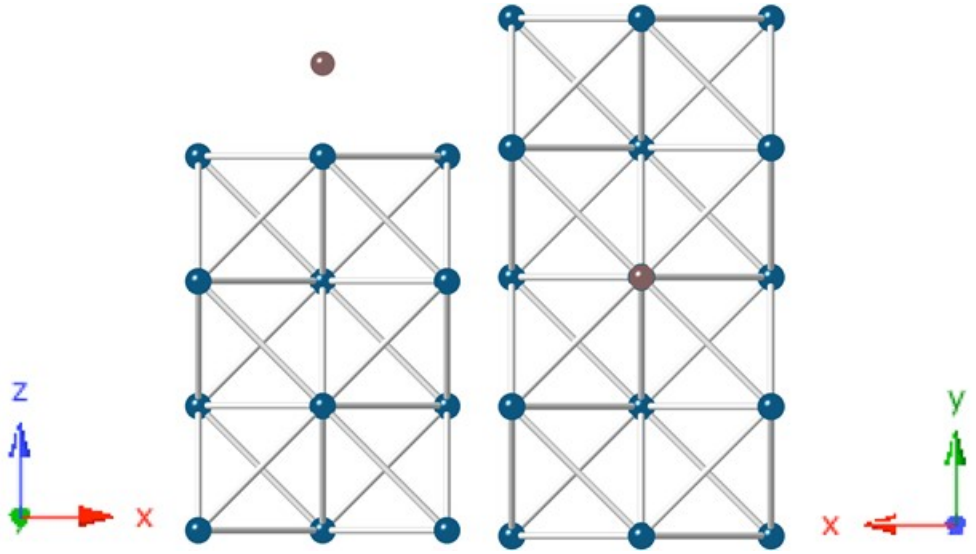

Fig. 3 Side and top-down view of a structure file generated by *oneAtomAdsorb*. The adsorbate (hydrogen atom, brown) adsorbed onto the top of a material (aluminum slab, blue representing aluminum atoms), specifically above a single atom.

3.3.3 Placing Adsorbates: Bridge Sites

The *twoAtomAdsorb* function takes the output list of *setUp*, with other settings, and returns a list of slabs with adsorbates as structure objects as the output. The adsorbate sites are located between 2 surface atoms as shown in Fig. 4. If the distance between 2 surface atoms is less than or equal to a given value, the center between those atoms is considered an adsorption site. The adsorbate can also be rotated with respect to the adsorbate site or the atom of the adsorbate meeting the slab surface. Two atoms of differing elevation can be considered part of the same surface. The function returns a list of unique structures as determined by the pymatgen[11] *StructureMatcher* tool.

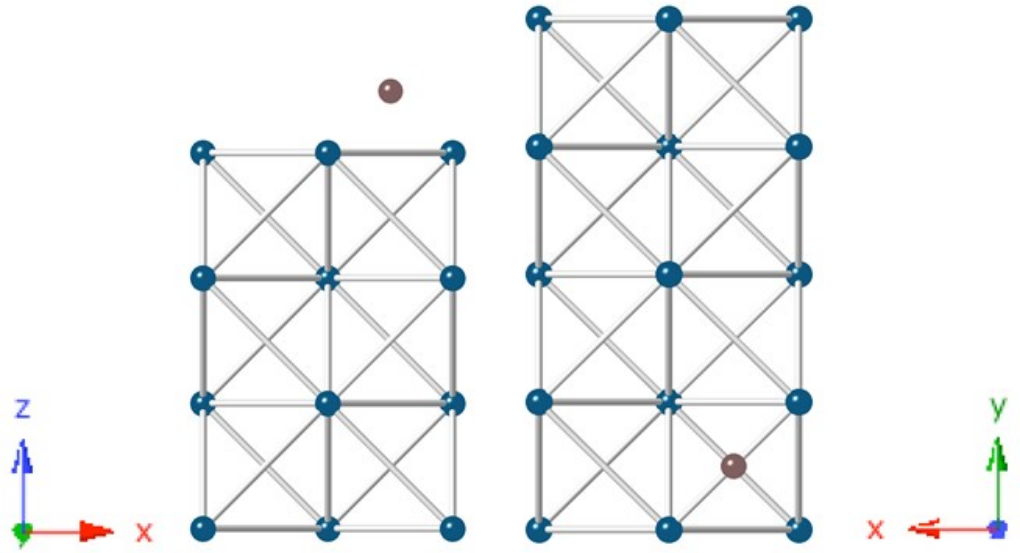

Fig. 4 Side and top-down view of a structure file generated by *twoAtomAdsorb*. The adsorbate (hydrogen atom, brown) adsorbed onto the top of a material (aluminum slab, blue representing aluminum atoms), specifically between 2 aluminum atoms.

3.3.4 Placing Adsorbates: Interstitial Sites

The *threeAtomAdsorb* function takes the output list of *setUp*, with other settings, and returns a list of slabs with adsorbates as structure objects as the output. The adsorbate is placed at the point center of the triangle formed by the 3 surface atoms as shown in Fig. 5. All translations and rotations are relative to this point. Three atoms of various elevation can be considered as part of the same surface. In order for the center point to be considered, both the area between the 3 surface atoms must be less than or equal to a given value and the angles formed between the 3 atoms must fall between a specified range of angles. The adsorbate can also be rotated with respect to the adsorption site or the atom of the adsorbate meeting the slab surface. The function returns a list of unique structures, as determined by the pymatgen[11] *StructureMatcher* tool.

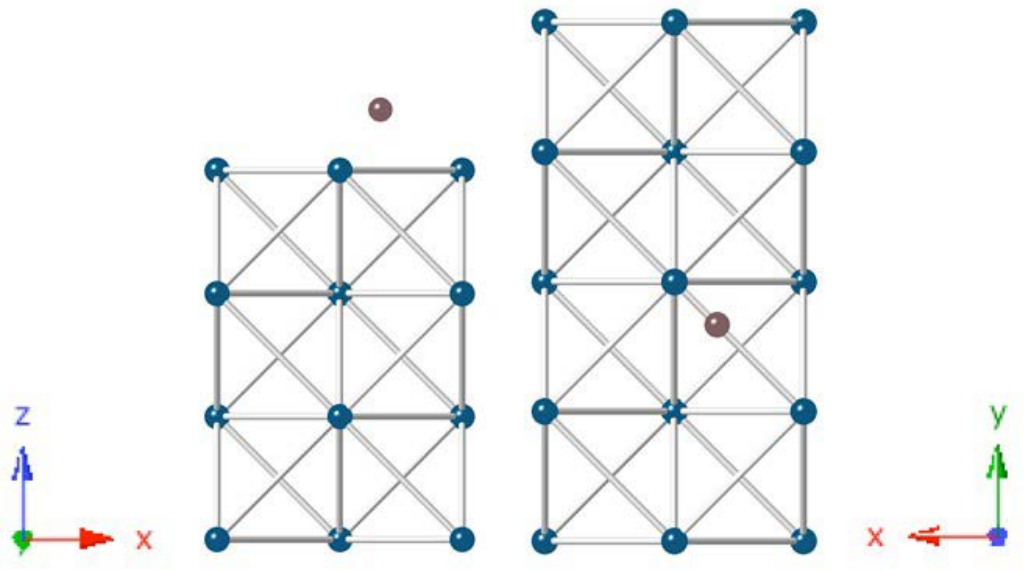

Fig. 5 Side and top-down view of a structure file generated by *threeAtomAdsorb*. The adsorbate (hydrogen atom, brown) adsorbed onto the top of a material (aluminum slab, blue representing aluminum atoms), specifically at the center of a triangle formed by 3 aluminum atoms.

3.3.5 Generating Adsorption Site List

The *writeStruct* function takes a list of structures as input and writes unique structures as POSCARS using the pymatgen[11] *StructureMatcher* tool. It generates unique directories in which to store these structures, starting from the name "1" and increasing incrementally from there.

This function is best used by taking the inputs of *one/two/threeAtomAdsorb*. It is recommended to use a single list as input, but a list of sublists containing structure objects can also be used as input. The function should be run when inside an orientation directory generated by *getGA_slabs* or *getMP_slabs* (example: path/to/parent/directory/Mg/111/). Typically, the slab inside the orientation directory will be used as input for *setup*, followed by using *one/two/threeAtomAdsorb*. Finally, *writeStruct* is used to generate and write the adsorbed structures inside the orientation directory, keeping all the slabs associated with the orientation in one directory.

3.4 Adsorption Analysis

The *orgDataAdsorb* function organizes data for slabs with adsorbates generated by the *makeLigSurface* functions. The function returns the adsorption energy results data in the file "resultsA.txt", which organizes adsorption energies based on material, surface orientation, and the site index. The code should be run in a

directory that contains slab subdirectories (i.e., 101, 100, and 111). When *orgDataAdsorb* is run, there must be a directory called "adsorbate" in the directory where this function is run that contains the adsorbate isolated in vacuum. To properly calculate the adsorption energy, the energy of both the slab in vacuum and the adsorbate in vacuum must be known. A single ionic step (i.e., a non-self-consistent) may be sufficient, though relaxation of the isolated molecule can be helpful.

This function can be run after adsorbates are placed on surfaces, with the recommended organization of adsorbed surfaces, then structurally optimized (refer to Appendix D.7). Energy values are read from the OSZICAR file, so any system that did not terminate on an ionic convergence will not appear in the "resultsA.txt" file. Data are separated based on the orientation and the adsorbate site, thus confusion between surfaces of differing materials are avoided. This function should be run in the same directory that *makeGA_slabs* or *makeMP_slabs* was initially run in.

4. Conclusions

This report details the high-throughput slab generation and molecular adsorption toolkit "adsorbates" that was developed as a part of the High Performance Computing Modernization Program's FY17 internship program under project number HIP-17-029. The toolkit was developed to aid in corrosion-resistant magnesium alloy design and uses high-fidelity DFT calculations to predict and evaluate the effect of potential secondary phases on the cathodic corrosion reaction thermodynamics. The framework consolidates available open source tools such as genetic algorithms, crystal structure databases, and slab generation tools in conjunction with a newly developed molecular adsorbate placement tool. The toolkit comprises 4 general stages: 1) generation of slabs from bulk structures that are user defined, generated from GA searches, or mined from existing databases; 2) calculating clean slab surface energies; 3) placing adsorbates on the slab surface based on nearest-neighbor configuration; and 4) calculating the binding energy of the adsorbate at each unique adsorption site. This framework can be used for corrosion investigation regardless of the materials system. In addition, it contains tools that can be used to improve any investigation involving adsorbates on a materials surface.

5. References

1. Fajardo S, Frankel GS. Effect of impurities on the enhanced catalytic activity for hydrogen evolution in high purity magnesium. Electrochimica Acta. 2015;165:255–267.
2. Südholz A, Kirkland N, Buchheit R, Birbilis N. Electrochemical properties of intermetallic phases and common impurity elements in magnesium alloys. Electrochemical and Solid-State Letters. 2011;14(2):C5–C7.
3. Birbilis N, Williams G, Gusieva K, Samaniego A, Gibson MA, McMurray HN. Poisoning the corrosion of magnesium. Electrochemistry Communications. 2013;34:295–298.
4. Limmer KR, Williams KS, Labukas JP, Andzelm JW. First principles modeling of cathodic reaction thermodynamics in dilute magnesium alloys. Corrosion. 2017;73(5):506–517.
5. Jain A, Ong SP, Hautier G, Chen W, Richards WD, Dacek S, Cholia S, Gunter D, Skinner D, Ceder G. Commentary: the materials project: a materials genome approach to accelerating materials innovation. Apl Materials. 2013;1(1):011002.
6. Tipton WW, Hennig RG. A grand canonical genetic algorithm for the prediction of multi-component phase diagrams and testing of empirical potentials. Journal of Physics: Condensed Matter. 2013;25(49):495401.
7. Sun W, Dacek ST, Ong SP, Hautier G, Jain A, Richards WD, Gamst AC, Persson KA, Ceder G. The thermodynamic scale of inorganic crystalline metastability. Science Advances. 2016;2(11):e1600225.
8. Kresse G, Furthmüller J. Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set. Physical Review B. 1996;54(16):11169–11186.
9. Giannozzi P, Baroni S, Bonini N, Calandra M, Car R, Cavazzoni C, Ceresoli D, Chiarotti GL, Cococcioni M, Dabo I. Quantum espresso: a modular and open-source software project for quantum simulations of materials. Journal of physics: Condensed Matter. 2009;21(39):395502.
10. Tipton W, Hennig R. Gainesville (FL): University of Florida. GASP: The genetic algorithm for structure and phase prediction [updated 2014 Apr 25; accessed 2017 Nov 2]. http://www.mse.ufl.edu/wp-content/uploads/gasp_manual.pdf.

11. Ong SP, Richards WD, Jain A, Hautier G, Kocher M, Cholia S, Gunter D, Chevrier VL, Persson KA, Ceder G. Python materials genomics (pymatgen): a robust, open-source python library for materials analysis. Computational Materials Science. 2013;68:314–319.

12. Mathew K, Singh AK, Gabriel JJ, Choudhary K, Sinnott SB, Davydov AV, Tavazza F, Hennig RG. MPInterfaces: a materials project based Python tool for high-throughput computational screening of interfacial systems. Computational Materials Science. 2016;122:183–190.

INTENTIONALLY LEFT BLANK.

Appendix A. Software Installation in a Virtual Environment

Prior to installing software, it is recommended to create a "software" directory in your home directory to store the virtual environment, MPInterfaces, and all other software. This helps prevent clutter in the home directory. Once a software has been installed, moving the location of the software will likely break the software. This is because the paths that were defined during installation are different and the original paths do not point to the new location of the software.

The first step in performing high-throughput work is connecting to a supercomputer. In order to use a great deal of the software available, access to root files of many programs is required. Because supercomputer infrastructures vary and root file permissions are not generally allowed, a virtual environment may be used to circumvent this issue. These environments provide root files that can be accessed and altered by the user as well as a space isolated from the main file structure; thus, they are ideal for use on shared clusters/networks. However, they are also useful for personal computers. Software packages can have conflicting dependencies, and installing several in one environment can cause problems. Creating separate virtual environments allows the packages to be installed independently from each other, and thus do not cause conflict.

A.1 Installing a Virtual Environment

There are several kinds of virtual environments, but Miniconda is recommended. The software can be downloaded at https://conda.io/miniconda.html by selecting the Python 3.6 Linux 64-bit bash installer package. In order to be fully compatible with the other tools in this suite, the 3.6 version of Python is required. To install Miniconda, copy this file to the computer/supercomputer, and then unpack and install with the following command:

```
bash [name of downloaded miniconda file]
```

To create the virtual environment, navigate to the file location, and enter the command

```
conda create --name [desired name of environment]
anaconda python=3.6
```

To activate the virtual environment, use the command

```
source activate [desired name of environment]
```

To exit the virtual environment, use the command

```
source deactivate
```

A.2 Installing Dependent Software Packages

Once the virtual environment has been activated, the "pip" command can be used to install several pieces of software. "pip" is used to access an online repository of software, from which one can obtain software packages and easily install them. To begin, enter the command

```
pip install numpy
```

This package contains several mathematical functions and transformations, and is required for the next package pymatgen. This package is also installed using pip with the command

```
pip install pymatgen
```

Finally, the software package MPInterfaces is required. This package has dependencies on both numpy and pymatgen, along with some others. These additional packages are installed along with MPInterfaces, so additional steps are not required. To obtain the package, navigate to the site https://github.com/henniggroup/MPInterfaces, click "clone or download", and finally "Download ZIP". Transfer this zip file to the relevant computer or cluster and unzip the file using

```
Unzip [name of file]
```

Following, enter the directory created and enter the command

```
python setup.py develop
```

An addition should be made to MPInterfaces-master/mpinterfaces/interface.py. *cover_surface2*. This modification of the Interface Class is presented in Appendix G. The standard *cover_surface* function is unable to rotate molecules relative to the location of adsorption. *cover_surface2* uses the *SymmOp* function to perform this transformation to the adsorbate, which also requires an additional 2 variables to define the rotation of the molecule relative to itself and relative to the adsorption site.

A.3 Importing Adsorbates.py Functions

Once the previous software has been installed, *adsorbates.py* can be used. This is started by placing the *adsorbates.py* file in the directory from which the functions should be run. Once this is done, the functions can be used in a python script through the line:

```
import adsorbates
```

All functions can then be used by typing the name of the function, and prefacing it with “adsorbates”. For example,

```
adsorbates.makeConvergeSlabs(…)
```

Any functions within the *makeLigSurface* function should be called as follows:

```
Adsorbates.makeLigSurface.oneAtomAdsorb(…)
```

Importing the *adsorbates.py* software will generate a directory called “__pycache__”, which contains files that speed up the initialization of the *adsorbates.py* package. We recommend avoiding this by adding “-B’ to the command line when running any python script that imports the package, as follows:

```
python –B example.py
```

Appendix B. Genetic Algorithm for Structure Prediction (GASP) Input Files

B.1 Installing GASP

There are 2 forms of the genetic algorithm for structure prediction (GASP) [1]: GASP and GASP-python. The former primarily uses bash while the latter uses solely python, although the operation is generally the same. GASP-python also has additional changes to improve efficiency and is recommended for its enhanced compatibility with the python framework developed here. GASP-python can be installed from https://github.com/henniggroup/GASP-python. Installation should be done within the python 3.6 virtual environment.

GASP may be used with several engines; however, the description here pertains only to the use of Vienna ab initio simulation package (VASP). To run GASP, a directory in the home directory called "bin" must contain 2 files: "callvasp" and "submit_GA". Because "callvasp" was written for a Slurm submission system, whereas most Department of Defense (DOD) supercomputers use the Portable Batch System submission system, the file will need to be edited to run properly. The file "submit_GA" is used as the submission file for the simulations in the GASP runs and will need to be customized in accordance with the submission system of the supercomputer being used.

GASP runs are initiated from within a directory named of the user's choosing that contains the relevant information for the GASP simulation. This directory, for example, "gasprun1", must include a file defining the GASP parameters as well as subdirectory named "inputs" that contains the VASP parameters files.

The GASP parameter file, "ga_input.yaml", can be customized to adjust the manner in which the genetic algorithm (GA) searches the computational space. However, most default settings intended for exploring the energy landscape will work well for this application. Settings that should be customized are based on the computational limits and expectations: CompositionSpace, EnergyCode, InitialPopulation, NumCalcsAtOnce, Constraints, and StoppingCriteria. Details on GASP settings and options in this file are located in the GASP user manual[1] and at https://github.com/henniggroup/GASP-python/blob/master/docs/usage.md.

The directory "/gasprun1/inputs" must contain 2 files that pertain to the VASP settings used in the simulations: INCAR and KPOINTS. KPOINTS should be an automatic mesh, and INCAR should be universal enough that it can be applied to any potential resulting structure from the GA. Example settings used in INCAR and KPOINTS are given in Appendix A. The directory "/gasprun1/inputs" may also

[1] Tipton WW, Hennig RG. A grand canonical genetic algorithm for the prediction of multi-component phase diagrams and testing of empirical potentials. Journal of Physics: Condensed Matter. 2013;25(49):495401.

contain an optional subdirectory “structs”. Any structures that are desired to seed the algorithm should be placed in the “structs” directory, written in a POSCAR format.

B.2 GASP Files

Example GASP input files have been included to guide the user in the settings recommended for the GA run for this application space. Additional details on the specific parameters and additional options may be found in the GASP manual[1]. After completing the GASP installation, it is recommended to perform a short VASP-based GASP run with a small initial population size and a static VASP calculation to make sure the pathways and job submissions are all correct before submitting the larger GA run.

B.2.1 VASP Submission Files

“Callvasp” and “qsubandpoll” files are required by GASP to perform VASP calculations. The “callvasp” file is used by GASP to start a VASP run, which calls “qsubandpoll” to submit and monitor VASP jobs within a given GASP run. Example versions of “callvasp” and “qsubandpoll” are available in the GASP manual and must be altered to match the high-performance computing specific queuing system and structure.

B.2.2 GASP Input

The input file for GASP called “GA_input” is used to set genetic algorithm parameters. It is recommended to test the GASP installation with a limited population size and a static VASP calculation to ensure the syntax is correct before submitting a full GA run. For exploring the existence of secondary phases, the default settings in GASP are recommended. An example GA_input file is provided as follows:

```
runTitle mgas

saveStateEachIter true

verbosity 4

popSize 20

InitialPopulation 30 random givenVol 20

InitialPopulation 2 poscars refstates/

compositionSpace 2 Mg As
```

```
Promotion 1

Variation1 0.2 0.2 structureMut 0 0.3 0.1

Variation2 0.80 0.8 Slicer 0.5 1 1 0.05 0 0

ObjectiveFunction pd vasp true KPOINTS INCAR Mg
POTCAR_Mg As POTCAR_As

Selection probDist 13 1

useNiggliReducedCell true

optimizeDensity 0.5 4

useRedundancyGuard both 0.1 0.1 0.1

ConvergenceCriterion maxNumGens 20

minInteratomicDistance 1.5

maxLatticeLength 35

minLatticeLength 1

maxLatticeAngle 140

minLatticeAngle 40

maxNumAtoms 16

minNumAtoms 2

minNumSpecies 2

doNonnegativityConstraint false

parallelize 20 18
```

Performing GASP simulations using VASP as the engine driver additionally requires VASP parameter files (INCAR, POTCAR, KPOINTS, and POSCAR) to be made available to perform the VASP calculations. Examples of recommended VASP settings are provided in Appendix C.

Appendix C. Vienna ab initio Simulation Package (VASP) Files

C.1 Vienna ab initio Simulation Package (VASP) Input Files

C.1.1 INCAR

The INCAR file is what is used to define the settings of the VASP simulation. However, one should take note of any materials that may have dipoles within them. If a dipole does appear, then the slab will not converge and the surface energy values will continuously rise during convergence. This can be corrected by adding the tags IDIPOL=3 and LDIPOL=TRUE. Details on these settings can be found on the VASP manual (https://cms.mpi.univie.ac.at/vasp/vasp/vasp.html).

ENCUT should be set to 140% of the largest ENMAX in the POTCAR directory. EDIFF of 1E-4 is accuracy to the 0.1 meV, which is sufficient for calculating surface energies. When calculating adsorption energies, higher levels of accuracy should be considered. For slabs, an example INCAR is as follows:

Example INCAR:

```
ENCUT=500
EDIFF=1E-4
PREC=Accurate
IBRION=2
ISIF=3
ISMEAR=1
SIGMA=0.1
NSW=50
```

C.1.2 KPOINTS

The KPOINTS file can be created using the script *reciprocal*, provided in Appendix G. There are 4 variables for this script: makeKPOINTS(Boolean, string, int, string). These variables are, respectively, whether the monolayer is a slab (True if slab), the type of mesh desired *(adsorbates.py* was written for "Gamma"), how detailed the mesh should be (in units of inverse Angstroms), and the path to which the POSCAR is and KPOINTS should be written. The KPOINTS file[1] generated is the explicitly written results of an automatic mesh.

[1] KPOINTs file: automatic k-mesh generation. Vienna (Austria): University of Vienna [accessed 2017 Nov 2]. https://cms.mpi.univie.ac.at/vasp/vasp/Automatic_k_mesh_generation.html.

The reason to use *reciprocal.py* instead of an automatic mesh is that slabs do not need multiple KPOINTS in the direction with vacuum spacing. Rather, only a single KPOINT is needed, which saves a great deal of calculation time. Thus, this script explicitly writes out what will be the same results as an automatic mesh, but reduces the number of KPOINTS in the z-direction if the Boolean is set to "True". If the material is not a slab, an automatic mesh KPOINTS file is recommended.

C.1.3 POTCAR

The VASP pseudopotential files, POTCAR, should be located in a common directory on each high performance computing system and can be copied as needed. For example, on Thunder, POTCAR files are located at /app/vaspapp/Potentials/. It is recommended to use PBE-5.2. When working with multicomponent systems, all relevant POTCAR files must be concatenated. To concatenate the POTCAR files, use the cat function (cat [path to file 1] [path to file 2] [path to file 3] ….. > [path to write new file to]).

C.1.4 POSCAR

The VASP structure file, POSCAR, contains the lattice parameters and angles as well as coordinates and type of each atom in the structure being simulated. These files can be created by the user, generated through *adsorbates*, or obtained through databases, such as Materials Project.

C.2 Creating a VASP "noz" Binary

When running VASP simulations on systems with vacuum spacing, a new binary must be used because a standard VASP binary will compress any vacuum spacing in a system over time. This not only can cause self-interaction in the slab, but also increase computation time. To solve this, the following should be included in the file "constr_cell_relax.F" before compiling the VASP binary. This file should contain the following:

```
SUBROUTINE CONSTR_CELL_RELAX(FCELL)
USE prec
REAL(q) FCELL(3,3)
!    just one simple example
!    relaxation in x and y directions only
SAVEX=FCELL(1,1)
SAVEY=FCELL(2,2)
```

```
FCELL=0  ! F90 style: set the whole array to zero
FCELL(1,1)=SAVEX
FCELL(2,2)=SAVEY
!    relaxation in z direction only
!    SAVE=FCELL(3,3)
!    FCELL=0  ! F90 style: set the whole array to
zero
!    FCELL(3,3)=SAVE
RETURN
END SUBROUTINE
```

Vacuum spacing is assumed to be only in the z-direction, but other directions can have their relaxation prevented as well (to do this, simply change which direction information is saved). Shearing of the x-y plane will also be prevented by disallowing relaxation in the z-direction. The file “constr_cell_relax.f90” contains the same information and may need to be included when compiling.

Appendix D. Adsorbates.py Software Variable Details

This appendix describes the variables that act as inputs for the functions in the adsorbates software package and the details of how each functions works, (i.e., the logic of each function). The details are for the functions in *adsorbates.py*. The order of the inputs listed for each function is the required order. If they are not entered in this way, the function will not run properly. In addition, if the variable has a name associated with it, then there is an assumed default value that can be changed by the user. For example, *setUp*(…) and *setUp*(…, from_slab=True) indicates that the input structure is a slab, while *setUp*(…, from_slab=False) indicates that the input POSCAR is not a slab. Such variables are noted in this appendix.

D.1 makeConvergeSlabs

```
makeConvergeSlabs(string, list, [x,y,z],
numIntervals=int, spaceIntervals=int)
```

- string: the path to the POSCAR to make slabs out of
- list: the list of desired vacuum paddings (Angstrom)
- [x,y,z]: the hkl orientation to cut along
- numIntervals: how many intervals to have for slab thickness
- spaceIntervals: how many thicknesses to skip between each interval

When generating the slab, the initial vacuum thickness is 1, with a vacuum padding of X-1 being added later. This is because, during slab generation, the vacuum is sometimes added in discrete units. By adding the vacuum padding in a separate step, the value closest to the desired thickness is reached. A slab with a vacuum padding of 0 can be generated, but this often causes the resulting unit cell to be unphysical by combining the top and bottom atomic layers of the slab into a single atomic layer. This combination into a single layer makes differentiating the atoms belonging to each surface impossible. “spaceIntervals” defines how many intervals to skip before generating a new slab, where only the slabs written are considered for “numInterval”. For example, if “spaceIntervals” is 2, only every other thickness is written up to “numInterval”.

To write the files, directories with the names formatted as “[slab thickness]_[vacuum thickness] are generated, with the corresponding slab being written in a POSCAR format in said directory.

D.2 makeMP_slabs

```
makeMP_slabs(int[, settings=list])
```

This function has 3 options for running. Regardless, it always requires an "int" to run, with this "int" representing the highest index for the Miller indices that the slabs will be generated along. A list is allowed as a second variable, when called through "settings=[contents of list]" after the initial "int" value. This list contains 10 variables as described in Table D-1.

Table D-1 Available settings for *makeMP_slabs*

Name	Description
elements (string)	A string of the elemental symbols corresponding to the desired elements to search over, separated with a "-"
eleList (list)	A list of elemental symbols corresponding to the desired elements to search over
mp_id (string)	The Materials Project application program interface (API) key ID. Should be unique to each user. Can be obtained by creating a free account on MaterialsProject.org and generating an API key.
maxTherm (int/float)	The maximum distance of each compound from the thermodynamic hull, in meV
min_thick (int/float)	The minimum thickness for each slab
min_vac (int/float)	The minimum vacuum spacing for each slab
include (string)	True or False. Whether or not to include the end point compositions in the slab generation.
maxAtom (int)	The maximum number of atoms allowed in the bulk POSCARs.
includeHulls (string)	True or False. Whether to include the maxAtom restriction on hull compounds.
skips (list)	A list of compounds to skip when making slabs. The list should be composed of strings that represent the name of the compound to skip.

If no list is given for settings, then the function requests values through input commands. The function states the variable name, and the user then enters the variable value and presses return/enter, leading to another variable being requested. Once all variables have been entered, the code will begin to run.

Like *makeGA_slabs*, this function creates a directory with the name of the Materials Project ID corresponding to each compound found. Within this directory will be subdirectories for the bulk compound and the slabs generated, with the corresponding POSCARs located in each directory.

A Materials Project application program interface (API) key is required for each user to obtain the structure files from Materials Project. This API key can be generated by accessing www.materialsproject.org, signing up for an account if not already logged in, and clicking "API" in the upper left banner. Under the section "API keys", there will be a generated key or an option to generate a key. This key

should be used for personal use only. The other settings are largely self-explanatory. The only detail to mention is that, if there are 2 materials with the same formula, the compound with a lower distance from the hull will be used.

D.3 makeGA_slabs

```
makeGA_slabs(string int/float, int/float, int)
```

- string: the path to the POSCAR files identified in the first list
- int/float: the minimum slab thickness to use
- int/float: the minimum vacuum padding to use
- int: the maximum value the indexes used in Miller indices can be

The algorithm takes each string in the list and generates a directory with the name of the string. It then creates a subdirectory named "bulk", in which the original POSCAR is copied. MPInterfaces is then used to generate slab structure objects for each potential surface up to the limit placed by the final "int" value. These slabs are compiled in a list, which pymatgen[1] sorts into a list of sublists. Each sublist contains structures that are symmetrically identical. A single slab from each group is written as a POSCAR file in a subdirectory that shares the name of the orientation the slab is derived from. For example, if the (111) surface is used, a subdirectory called "111" will be created and a POSCAR file representing the slab is written within it and called "POSCAR". The function attempts to use orientation names that do not contain negative numbers, but this is not always feasible.

This script is ideal for structures identified using a genetic algorithm, though any bulk crystal can be used. Some directories will begin with a '-', which can be difficult for the *os* package to parse. To circumvent this, add a './' prior to the '-' for all leading negative signs. For example, to change into the directory "-110", one would write the line "os.chdir('./-110')".

D.4 addsd

```
addsd(string[, int/float])
```

- string: path to the POSCAR to add selective dynamics to

[1] Ong SP, Richards WD, Jain A, Hautier G, Kocher M, Cholia S, Gunter D, Chevrier VL, Persson KA, Ceder G. Python materials genomics (pymatgen): a robust, open-source python library for materials analysis. Computational Materials Science. 2013;68:314–319.

- int/float: the fractional amount to shift the selective dynamics cutoff point. Assumed to be 0 unless given. Positive values increase the number of frozen atoms and negative values decrease the number.

This *addsd* function adds selective dynamics to a structure. Selective dynamics is the option to restrict the ability of some atoms to move during relaxation. This can be turned on by line 7 of a POSCAR beginning with an "S". What follows is irrelevant. All lines below line 6 are shifted down by one line to accommodate. This function will freeze the position of all atoms below the halfway point of a material. If there is an adsorbate on a slab, the middle point will be defined between the bottom of the slab and the top of the adsorbate.

The advantage of selective dynamics is to decrease computation time. Fewer atoms able to move results in fewer calculations, and thus fewer resources used during calculation.

D.5 getAngles

```
getAngles([x,y,z], [x,y,z])
```

This function takes 1 × 3 lists representing different vectors as inputs, and returns the angle (in degrees) between those vectors as output. It acts as a helper function for *makeLigSurface*.

D.6 getAbsCoord

```
getAbsCoord([x, y, z], int/float, int/float,
int/float)
```

This function takes a 1 × 3 list and 3 numbers as input, and returns a 1 × 3 array as an output. This is used to obtain the locations of atoms in a standard x-y-z coordinate system. It acts as a helper function to *makeLigSurface*.

D.7 makeLigSurface

```
makeLigSurface()
```

This class contains 5 functions, which are detailed in the following subsections. The purpose of the class is to place adsorbates on the surface of slabs. The first function sets up the adsorbate calculations, the following 3 functions place adsorbates on a slab (on a single atom, between 2 atoms, and in the middle of 3 atoms), and the fifth function eliminates structurally symmetric configurations and writes the structure files.

D.7.1 setUp

```
setUp(string, string, int/float, int/float, [x,
y, z], minSurfArea=int/float, from_slab=Boolean,
hkl=[x, y, z], alwaysEqual=Boolean)
```

- string: the path to the input structure POSCAR
- string: the path to the adsorbate POSCAR
- int/float: the minimum slab thickness
- int/float: the minimum vacuum padding
- [x,y,z]: the supercell to use in the slab generation
- minSurfArea: the minimum surface area of the slab surface (Angstrom2)
- from_slab: whether the input structure is a slab (True) or not (False)
- hkl: the Miller index to make a slab along
- alwaysEqual: whether the a and b lattice vectors should be equal while increasing the surface area (True) or can vary (False)

This function is designed to set up the future calculations for adsorption placement. It begins by generating a slab from the input structure (or simply creating a structure object if from_slab=True and hkl=[0,0,1]), then continues by checking the area of the surface of the slab. If it is less than "minSufArea", then supercells of the slab are generated until the minimum surface area is reached.

The next step in the algorithm is to identify all atoms that exist in the top layer of the material. The top layer of the material is defined as any atom that has no other atoms directly above it. In the algorithm, one atom is "above" another if they form a vector parallel to the slab surface normal. To determine this, a vector is formed between the 2 atoms and compared to a vector perpendicular to the surface of the slab (identified by taking the cross product of the *a* and *b* vectors of the unit cell). If the vectors form a dot product that is 0.98 or greater, the atom that is higher along the *c* vector is put under consideration for being a top atom.

To account for the possibility that 2 atoms are above each other, but do not appear to be based on the original bulk fractional coordinates, transformations of one atom along the *a* and *b* vectors are also considered. If one atom is found to be above the other after a transformation, then it is put under consideration for being a top atom. If an atom was previously under consideration, but another atom was found to be

"above" it, the first atom is removed from consideration while the second is added for consideration.

This function returns a list of 7 objects, 2 integers, a 1 × 3 vector, and a structure object. The details of the objects in the list are as follows:

- pymatgen[1] structure object slab
- list of atoms identified as being in the top layer
- list of all atomic positions
- list of elements in the slab
- float of the minimum thickness of the slab
- float of the minimum vacuum spacing
- adsorbate as a ligand object

The 2 integers are recommended values for the maximum height difference between atoms considered for adsorbate placement and the maximum area that can be formed between 3 atoms when considering adsorption, respectively. The independent structure object is the same slab but reoriented so the bottom surface is now the top surface. This is made available in case the user wishes to check if the surfaces are identical or if they are structurally/chemically different.

D.7.2 oneAtomAdsorb

```
oneAtomAdsorb(list, string, [x,y,z], int/float,
[x,y,z],int/float)
```

- list: the list output by the *setUp* function. See Section D.7.1 for details on what is in this list.
- string: the atomic symbol for the atom on the adsorbate that should be in contact with the surface
- [x,y,z]: the angles to rotate the ligand toward. x is rotation around the x-axis, y is rotation around the y-axis, and z is rotation around the z-axis. (degrees). The center of rotation is the adsorption site of the slab.
- int/float: the distance of the adsorbate from the adatom (Angstrom)
- [x,y,z]: the axis in which to rotate the molecule in position around. At least one of these values must be nonzero.

- int/float: the number of degrees to rotate the adsorbate (degrees). If 0, there will be no rotation about the adsorption point regardless of the values of the previous list.

This function takes the output list of *setUp*, with other settings, and returns a list of slabs with adsorbates as structure objects as the output. The adsorbate positions considered are only above atoms. The rotation and translation of the adsorbate is relative to this site.

D.7.3 twoAtomAdsorb

```
twoAtomAdsorb(list, int/float, string, [x,y,z],
int/float, int/float, [x,y,z], int/float)
```

- list: the list output by the *setUp* function. See Section D.7.1 for details on what is in this list.
- int/float: the maximum distance between 2 surface atoms considered for adsorption
- string: the atomic symbol for the atom on the adsorbate that should be in contact with the surface
- [x,y,z]: the angles to rotate the ligand toward. x is rotation around the x-axis, y is rotation around the y-axis, and z is rotation around the z-axis. (degrees). The center of rotation is the site of adsorption for the slab.
- int/float: the distance of the adsorbate from the adatom (Angstrom)
- int/float: the maximum vertical distance between the 2 surface atoms considered for adsorption (Angstrom)
- [x,y,z]: the direction in which to rotate the molecule in position, in Miller indices. The center of rotation is the atom of the adsorbate that meets the slab surface. Magnitude of the vector does not matter. At least one of the values must be nonzero.
- int/float: the number of degrees to rotate the adsorbate (degrees). If set to 0, no rotation will occur about the adsorption point regardless of the values of the previous list.

This function takes the output list of *setUp*, with other settings, and returns a list of slabs with adsorbates as structure objects as the output. If the distance between 2 surface atoms is less than or equal to a given value, the center between those atoms is considered an adsorption site. An additional restriction is that the atoms must be less than a maximum distance along the z-direction. *setUp* gives a value

(defaultHeight) that is the vertical distance between the lowest surface atom and the highest.

D.7.4 threeAtomAdsorb

```
threeAtomAdsorb(list, int/float, string, [x,y,z],
int/float, int/float, [x,y], [x,y,z], int/float)
```

- list: the list output by the *setUp* function. See Section D.7.1 for details on what is in this list.
- int/float: the maximum area between 3 surface atoms considered for adsorption
- string: the atomic symbol for the atom on the adsorbate that should be in contact with the surface
- [x,y,z]: the angles to rotate the ligand toward. x is rotation around the x-axis, y is rotation around the y-axis, and z is rotation around the z-axis. (degrees). The center of rotation is the site of adsorption.
- int/float: the distance of the adsorbate from the adatom (Angstrom)
- int/float: the maximum vertical distance between the 2 surface atoms considered for adsorption (Angstrom)
- [x,y]: the minimum and maximum angles that can form between the 3 atoms considered for adsorption (degrees)
- [x,y,z]: the direction in which to rotate the molecule in position, in Miller indices. The center of rotation is the atom of the adsorbate that meets the slab surface. Magnitude of the vector does not matter. At least one of these values has to be nonzero.
- int/float: the number of degrees to rotate the adsorbate (degrees). If set to 0, no rotation will occur about the adsorption point regardless of the values of the previous list.

This function takes the output list of *setUp*, with other settings, and returns a list of slabs with adsorbates as structure objects as the output. If the area between 2 surface atoms is less than or equal to a given value, the center of the area formed by these atoms is considered the site for adsorption. An additional restriction is that the angles formed between the 3 atoms must fall between a range of angles. Finally, the atoms considered must be less than a maximum distance along the z-direction. *setUp* gives a value (defaultHeight) that is the vertical distance between the lowest surface atom and the highest. It also recommends a maximum area (defaultArea)

that is calculated by taking the area of the slab surface and dividing by the number of surface atoms present in the material and multiplying by 3.

D.7.5 writeStructs

```
writeStructs(list)
```

- list: a list that either contains structure objects or contains sublists that contain structure objects

This function takes a list that contains structure objects or list of sublists containing structure objects. The function uses the pymatgen[1] *StructureMatcher* tool to organize the input structures. This requires a single list of structures as input. Thus, if a list of sublists is used, then all structures within the sublist are combined into a single list used in the *StructureMatcher* tool. The tool then creates a list of sublists, where each sublist contains identical structures. This function then takes the first structure in each sublist and writes them to a unique directory. The first unique structure is placed in a directory called "1", the second unique structure is placed in a directory called "2", and so on until one structure from each sublist has been written to.

D.8 orgDataConvergence

```
orgDataConvergence()
```

Organizes data for slabs generated with *makeConvergeSlabs*. The code should be run in the same directory where *makeConvergeSlabs* was run, because the assumed organization of structures is that generated by *makeConvergeSlabs*.

D.9 orgDataSlabs

```
orgDataSlabs()
```

Organizes data for slabs generated with *makeGA_slabs* and *makeMP_slabs*. The code should be run in the same directory where these functions were run because the assumed organization of structures is that generated by these functions.

D.10 orgDataAdsorb

```
orgDataAdsorb()
```

Organizes data for slabs generated with the *writeStruct* function of *makeLigSurface*. The code should be run in the same directory where *makeGA_slabs* or *makeMP_slabs* was run because the assumed organization of

structures is that generated by *writeStruct*, which is dependent on the organization of *makeGA_slabs* and *makeMP_slabs*.

INTENTIONALLY LEFT BLANK.

Appendix E. Python Script for Complete adsorbates.py Source

This appendix appears in its original form, without editorial change.

#Copyright (c) 2017 Joshua Thomas Paul
#MIT License

#Permission is hereby granted, free of charge, to any person obtaining a copy of
#this software and associated documentation files (the "Software"), to deal in
#the Software without restriction, including without limitation the rights to
#use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of
#the Software, and to permit persons to whom the Software is furnished to do so,
#subject to the following conditions:

#The above copyright notice and this permission notice shall be included in all
#copies or substantial portions of the Software.

#THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
#IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
#FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
#COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
#IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
#CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

```
from six.moves import range

import os
import sys
import math
import copy

import numpy as np

from pymatgen.core.structure import Structure, Molecule
from pymatgen.core.lattice import Lattice
from pymatgen.core.surface import Slab, SlabGenerator
from pymatgen.core.operations import SymmOp
from pymatgen.util.coord_utils import get_angle

from mpinterfaces.transformations import reduced_supercell_vectors
```

```
from mpinterfaces.utils import get_ase_slab, align_axis
from mpinterfaces.default_logger import get_default_logger

################################

from pymatgen.io.vasp.inputs import Poscar
from operator import itemgetter
from pymatgen.core.periodic_table import Element
from pymatgen.core.composition import Composition
from pymatgen.analysis.structure_matcher import StructureMatcher
from math import acos

from mpinterfaces.utils import align_axis, add_vacuum
from pymatgen.analysis.structure_matcher import StructureMatcher
from mpinterfaces.interface import Interface, Ligand
from pymatgen.matproj.rest import MPRester as mpr
def makeConvergeSlabs(struct, vacs, hkl,numIntervals=4,spaceIntervals=1):
   # To generate slabs, run this function in a
   # directory where the convergence to be tested
   # in. Give a bulk structure object (that you will converge),
   # list of vacuum spacings, and the hkl to make the
   # surface along.
   structure = Structure.from_file(struct)
   a,b,c = hkl
   iface_slab = Interface(structure, hkl=[a,b,c],
                  min_thick=.1, min_vac=1,
                  supercell=[1,1,1],
                  primitive=False,from_ase=True)

   coords = iface_slab.cart_coords
   zs = [z for [x,y,z] in coords]
   interval1 = max(zs)-min(zs)
   if interval1==0:
      minThick=.1
      while interval1==0:
         minThick+=.25
         iface_slab = Interface(structure, hkl=[a,b,c],
                  min_thick=minThick, min_vac=1,
                  supercell=[1,1,1],
                  primitive=False,from_ase=True)

         coords = iface_slab.cart_coords
         zs = [z for [x,y,z] in coords]
```

```
        interval1 = max(zs)-min(zs)
    iface_slab = Interface(structure, hkl=[a,b,c],
                min_thick=interval1*2*1.5, min_vac=1,
                supercell=[1,1,1],
                primitive=False,from_ase=True)
    coords = iface_slab.cart_coords
    zs = [z for [x,y,z] in coords]
    interval2 = max(zs)-min(zs)
    interval = interval2-interval1


    thicks = []
    thicks.append(round(interval1,2))
    for x in range(numIntervals):
        thicks.append(round(interval1+interval*(x+1)*spaceIntervals,2))



    for min_thick in thicks:  # Minimum thickness to make slabs. Note that they
should
                            # be in intervals of the smallest thickness possible, ie
                            # the height of the slab made using a single unit cell
        for min_vac in vacs: # Minimum vacuum spacing
            try:
                os.mkdir(str(min_thick)+'_'+str(min_vac))  # Makes a slab directory
            except:
                pass

            iface_slab = Interface(structure, hkl=hkl,
                        min_thick=min_thick, min_vac=1,
                        supercell=[1,1,1],
                        primitive=False,from_ase=True)
            iface_slab.sort()
            iface_slab = align_axis(iface_slab) # Align slab surface normal to the c
axis
            iface_slab = add_vacuum(iface_slab, min_vac-1) # Apply vacuum spacing
            iface_slab.to('poscar', str(min_thick)+'_'+str(min_vac)+'/POSCAR') #
Write file


def makeMP_slabs(index=1, settings=[]):

    if settings==[]:
```

```
        elements=''
        numEle = int(input('Enter number of elements | '))
        eleList = []
        for x in range(numEle):   # Loop as many times as elements to input
           addEle = input('Type element symbol | ')
           elements+=addEle
           if x!=numEle-1:
              elements+='-'
           eleList.append(addEle)
        mp_id = input('Enter MaterialsProject API key | ')  # Enter
MaterialsProject API key. Should be the one
                                        # specific to your account, not someone
else's
        maxTherm = float(input('Enter maximum distance from thermodynamic
hull (in meV) | '))
        min_thick = float(input('Enter minimum slab thickness (in angstrom) | '))
        min_vac = float(input('Enter minimum vacuum spacing (in angstrom) | '))
        include = input('Include endpoints? True or False | ')  # Include pure
compositions in MaterialsProject pull
        skips = [input('Skip any compounds? Separate elemental compositions
with a , and entered in all potential elemental orders as well. If no compounds
should be skipped, hit return  | ')] # Any compounds to be skipped over?
        maxAtom = int(input('What is the maximum number of atoms in the
compounds taken? | '))  # Upper limit on number of atoms
        includeHulls = input('Include this limitation on hull compounds?
True/False | ')


    elif len(settings)==10:
       elements, eleList, mp_id, maxTherm, min_thick, min_vac, include,
maxAtom, includeHulls, skips = settings
    else:
       print('Settings is not the appropriate length. Terminating')
       exit()

    comps = mpr(mp_id).get_data(elements) # Obtain all structures for the system
defined
                              # (does not include inputs)

    finalSet = {}
    record = {}
```

```
    # Obtain list of compounds

    for comp in comps:  #
        energy = comp['e_above_hull']   # Obtain energy above hull of compound
        if energy<=maxTherm*.001:
            if energy==0:
                if includeHulls=='True': # Continue if hulls should be restricted by
maxAtom
                    if comp['nsites']<=maxAtom:
                        formula = comp['pretty_formula']      # Obtain reduced formula (to
organize data)
                        finalSet[formula]=comp['material_id'] # Obtain material id (to later
pull structure)
                        record[formula]=comp['e_above_hull']  # Note energy above hull
(so most stable structure is selected)
                else:  # Continue if all hull compounds should be taken
                    formula = comp['pretty_formula']
                    if formula not in skips:
                        finalSet[formula]=comp['material_id']
                        record[formula]=comp['e_above_hull']
            else: # Loop over off hull compounds
                if comp['nsites']<=maxAtom:
                    formula = comp['pretty_formula']
                    if formula not in skips:
                        if formula not in record:
                            finalSet[formula]=comp['material_id']
                            record[formula]=comp['e_above_hull']
                        else:
                            i=2
                            while formula+'_'+str(i) in record:
                                i+=1
                            finalSet[formula+'_'+str(i)]=comp['material_id']
                            record[formula+'_'+str(i)]=comp['e_above_hull']
    for ele in eleList: # Add pure end point compounds to list
        soloComps = mpr(mp_id).get_data(ele)
        for comp in soloComps:
            if comp['e_above_hull']==0:
                formula = comp['pretty_formula']
                if formula not in skips:
                    finalSet[formula]=comp['material_id']

    os.mkdir(elements)
    os.chdir(elements)
```

```
    for comp in finalSet: # For each compound, create a bulk directory and slabs
        os.mkdir(comp)
        os.chdir(comp)
        strt = mpr(mp_id).get_structures(finalSet[comp])[0]
        os.mkdir('bulk')
        Poscar(strt).write_file('bulk/POSCAR')
        structs = []
        structs2= []
        results=[]
        for a in range(-1*index, index):
            for b in range(-1*index, index):
                for c in range(-1*index, index):
                    supercell = [1, 1, 1]
                    if a==0 and b==0 and c==0:
                        pass
                    else:
                        hkl = [a,b,c]

                        iface_slab = Interface(strt, hkl=hkl,
                                    min_thick=min_thick, min_vac=1,
                                    supercell=supercell,
                                    primitive=False,from_ase=True)
                        iface_slab.sort()
                        iface_slab = align_axis(iface_slab)

                        iface_slab = add_vacuum(iface_slab, min_vac-1)
                        structs2.append([iface_slab, str(a)+str(b)+str(c)])
                        structs.append(iface_slab)
        results = StructureMatcher().group_structures(structs,anonymous=False)
        print(len(results))
        counter = 1
        for slab in results:
            for s in list(reversed(structs2)):
                if len(StructureMatcher().group_structures([slab[0],s[0]]))==1:
                    print(s[1])
                    os.mkdir(str(s[1]))
                    slab[len(slab)-1].to('poscar',str(s[1])+'/POSCAR')
                    break
        os.chdir('../')


def makeGA_slabs(pathToCompList, min_thick, min_vac, index=1):
```

```
    # Will generate slabs from a list of POSCARS. The path to
    # a directory filled with the POSCARs to turn into slabs is
    # taken as the input. Afterward, the minimum thickness of these slabs and
    # minimum vacuum spacing are given. Optional is the maximum
    # miller index, which is assumed to be 1 if not given

    for comp in os.listdir(pathToCompList):
        os.mkdir(comp) # Make a directory for the crystal being made into slabs
        os.chdir(comp)
        os.mkdir('bulk') # Make a directory to calculate the bulk energy. To prevent
this, comment out this and the next line
        os.system('cp ../'+pathToCompList+'/'+comp+' bulk/POSCAR')
        strt = Structure.from_file('bulk/POSCAR')
        structs = []
        structs2= []
        for a in range(-1*index, index):        # A list of the indices to loop over
            for b in range(-1*index, index):    # A list of the indices to loop over
                for c in range(-1*index,index): # A list of the indices to loop over
                    supercell = [1, 1, 1] # Whether to make a supercell
                    if a==0 and b==0 and c==0: # Make sure that there is some indici cut
along
                        pass
                    else:
                        hkl = [a,b,c]

                        iface_slab = Interface(strt, hkl=hkl,             # Create slab
                                    min_thick=min_thick, min_vac=1,
                                    supercell=supercell,
                                    primitive=False,from_ase=True)
                        iface_slab.sort()       # Organize atoms so they are properly
grouped
                        iface_slab = align_axis(iface_slab)  # Align c axis so that slab
surface is parallel to AB plane

                        iface_slab = add_vacuum(iface_slab, min_vac-1)  # Add vacuum
spacing. -1 since 1 A was initially used in slab generation
                        structs2.append([iface_slab, str(a)+str(b)+str(c)])
                        structs.append(iface_slab)
        results = StructureMatcher().group_structures(structs,anonymous=False) #
Takes a list of structures as inputs, and returns
                                                    # a list of structure lists. Each
sublist is a grouping
```

```
                                                    # of symmetrically equivalent
structures
        counter = 1
        for slab in results:
            for s in list(reversed(structs2)):  # Reversed so positive indicies are
selected as names first
                if len(StructureMatcher().group_structures([slab[0],s[0]]))==1: # If only
one list is returned, the structures are identical
                    os.mkdir(str(s[1]))
                    slab[len(slab)-1].to('poscar',str(s[1])+'/POSCAR')
                    break
        os.chdir('../')


def addsd(structure,shift=0):
    lines = open(structure,'r').readlines()
    zs = []
    if lines[7][0]!='S':  # Checks whether the "S" tag is present yet
        for num in range(len(lines)):
            if num >7:
                zs.append(float(lines[num].split()[2])) # Adds all z coordinates to a list
        mid = (max(zs)+min(zs))/2 # Finds the center of the slab
        with open(structure,'w') as f:
            for num in range(len(lines)):
                if num>7:
                    splits = lines[num].split()
                    if float(splits[2])>mid+shift: # If atom above the center, allow
movement
                        f.write(splits[0]+' '+splits[1]+' '+splits[2]+' T T T ')
                    else: # If atom not above center, fix position
                        f.write(splits[0]+' '+splits[1]+' '+splits[2]+' F F F ')
                    if len(splits)>3: # If the atom had an identifer (ex: '.5 .5 .5 Mg'), add
the identifier
                        f.write(splits[3]+' \n')
                    else:
                        f.write(' \n')
                elif num<7: # Write all lines prior to line 7 as is
                    f.write(lines[num])
                elif num==7:
                    f.write('S \n') # Add 'S' tag whil keeping
                    f.write(lines[num]) # Add previous identifier
        os.chdir('../')
    else: # Continue if 'S' tag was present
```

```
        for num in range(len(lines)):
            if num >8:
                zs.append(float(lines[num].split()[2]))  # Adds all z coordinates to a list
        mid = (max(zs)+min(zs))/2 # Finds center of the slab
        with open(structure,'w') as f:
            for num in range(len(lines)):
                if num>8:
                    splits = lines[num].split()
                    if float(splits[2])>mid+shift: # If atom above the center, allow
movement
                        f.write(splits[0]+' '+splits[1]+' '+splits[2]+' T T T ')
                    else: # If atom not above center, fix position
                        f.write(splits[0]+' '+splits[1]+' '+splits[2]+' F F F ')
                    if len(splits)>3: # If the atom had an identifer (ex: '.5 .5 .5 Mg'), add
the identifier
                        f.write(splits[3]+' \n')
                    else:
                        f.write(' \n')
                else:
                    f.write(lines[num]) # Write all lines prior to atomic positions

def getAngles(array1,array2):
    mag1 = np.linalg.norm(array1)
    mag2 = np.linalg.norm(array2)
    dotted = np.dot(array1,array2)
    if float(dotted/mag1/mag2)>=float(1.0):
        angle = 0
    else:
        angle = acos((dotted/mag1/mag2))
    return angle*180/np.pi



def getAbsCoord(site,a,b,c):
    return np.array([site.a*a[0]+site.b*b[0]+site.c*c[0], \
                     site.a*a[1]+site.b*b[1]+site.c*c[1], \
                     site.a*a[2]+site.b*b[2]+site.c*c[2]])

class makeLigSurface():

    def setUp(slabName, adsorbName,min_thick,min_vac,supercell,
minSurfArea=0.1,from_slab=True,hkl=[0,0,1],alwaysEqual=False):
```

```
    # slabName: Path to/name of the POSCAR file containing the slab or to-be
slab
    # adsorbName: Path to/name of the POSCAR file containing the adsorbate
    # min_thick: Minimum thickness of slab
    # min_vac: Minimum vacuum padding in the slab
    # supercell: Supercell to start from in adsorbate generation
    # minSurfArea: The minimum surface area of the slab
    # from_slab: Whether the POSCAR 'slabName' is a slab or not. Default to
True
    # hkl: The hkl indice to cut along. By default [0 0 1]
    # alwaysEqual: During surface area comparisons, whether or not to always
increase the
    #           supercell in the x and y direction simultaneously, or alternating.
Default to alternating (False)

    strt_start = Structure.from_file(slabName)
    mol_struct = Structure.from_file(adsorbName)
    mol = Molecule(mol_struct.species, mol_struct.cart_coords)
    ligand = Ligand([mol]) # Turn adsorbate into object

    strt = Interface(strt_start, hkl=hkl,
                min_thick=min_thick, min_vac=min_vac,
                supercell=supercell,
                primitive=False, from_ase=True,start_from_slab=from_slab)
    strt_novac = Interface(strt_start, hkl=hkl,
                min_thick=min_thick, min_vac=0,
                supercell=supercell,
                primitive=False,from_ase=True,start_from_slab=from_slab)
    strt_novac.sort()
    strt_novac = align_axis(strt_novac)
    strt_novac.to('poscar','POSCAR_slab_noVac.vasp')
    strt_novac.to('poscar','POSCAR_slab_noVac_reduced.vasp')


    # Get the list of sites in the slab. Then get a list [keep]
    # of atoms that have no other atom above it along the z-axis
    strt.sort()
    strt = align_axis(strt)
    lines = open('POSCAR_slab_noVac.vasp','r').readlines()
    a = [float(num) for num in lines[2].split()]
    b = [float(num) for num in lines[3].split()]
    c = [float(num) for num in lines[4].split()]
```

```
    # Scale the supercell size until it reaches the specified
    # minimum surface area. If alwaysBoth=False, increases
    # supercell to keep lattice vector sizes as similar as possible.
    if np.linalg.norm(np.cross((a),(b)))<minSurfArea*.9:
       aMultiple = 1
       bMultiple = 1
       if alwaysEqual==False: # If False, scale vectors so they are about equal,
                              # then increase the size of lattice vector a
          while
aMultiple*bMultiple*np.linalg.norm(np.cross((a),(b)))<minSurfArea*.9:
             if aMultiple*np.linalg.norm(a)>bMultiple*np.linalg.norm(b):
                supercell[1]+=1
                bMultiple+=1
             elif aMultiple*np.linalg.norm(a)<bMultiple*np.linalg.norm(b):
                supercell[0]+=1
                aMultiple+=1
             else:
                supercell[0]+=1
                aMultiple+=1
       else: # If True, scale lattice vectors so they are approximately equal, then
             # create supercell in x and y directions simultaneously until
minSurfArea
             # is reached
          if aMultiple*np.linalg.norm(a)*.9>bMultiple*np.linalg.norm(b):
             while aMultiple*np.linalg.norm(a)*.9>bMultiple*np.linalg.norm(b):
                supercell[1]+=1
                bMultiple+=1
          elif aMultiple*np.linalg.norm(a)<.9*bMultiple*np.linalg.norm(b):
             while aMultiple*np.linalg.nor(a)<.9*nbMultiple*p.linalg.norm(b):
                supercell[0]+=1
                aMultiple+=1
          while
aMultiple*bMultiple*np.linalg.norm(np.cross((a),(b)))<minSurfArea*.9:
             supercell[0]+=1
             supercell[1]+=1
             aMultiple+=1
             bMultiple+=1


    strt = Interface(strt_start, hkl=hkl,
                     min_thick=min_thick, min_vac=min_vac,
                     supercell=supercell,
                     primitive=False, from_ase=True,start_from_slab=True)
```

```
strt_novac = Interface(strt_start, hkl=hkl,
                min_thick=min_thick, min_vac=0,
                supercell=supercell,
                primitive=False, from_ase=True,start_from_slab=True)
strt_novac.sort()
strt_novac = align_axis(strt_novac)
strt_novac.to('poscar','POSCAR_slab_noVac.vasp')
strt_novac.to('poscar','POSCAR_slab_noVac_reduced.vasp')

# get the list of sites in the slab. Then get a list [keep]
# of atoms that have no other atom above it along the z-axis
strt.sort()
strt = align_axis(strt)
lines = open('POSCAR_slab_noVac.vasp','r').readlines()
a = [float(num) for num in lines[2].split()]
b = [float(num) for num in lines[3].split()]
c = [float(num) for num in lines[4].split()]

strt.to('poscar','POSCAR_slab.vasp')

lines = open('POSCAR_slab.vasp','r').readlines()
i = 0
a_vec = [float(num) for num in lines[2].split()]
b_vec = [float(num) for num in lines[3].split()]
c_vec = [float(num) for num in lines[4].split()]
aMag = np.linalg.norm(a_vec)
bMag = np.linalg.norm(b_vec)
cMag = np.linalg.norm(c_vec)
sites = strt.cart_coords
normal = np.cross(a_vec, b_vec)
xShift = np.array(a_vec)
yShift = np.array(b_vec)

# Begin identifying sites that correspond to the top surface atoms.
# This is defined as no atoms being above the site in the direction
# normal to the surface. The "_Shift" variables are present to
# take into account that fractional coordinates may not identify
# one site as being above an atom due to unit cell shape, but when
# supercells are created it clearly is.
keep = []
kept = []
index = 0
maxZs = {}
```

```
    loop=0
    while loop<4:
       index=0
       for site in strt.sites:
          x,y,z = getAbsCoord(site,a_vec,b_vec,c_vec)
          addIt = True
          breakAgain = False
          if keep != []:
             for atom in keep:
                   x2,y2,z2, oldIndex = atom
                   atomVectors = []
                   atomVectors.append(np.array([x-x2,y-y2,z-z2]))
                   atomVectors.append(np.array([x-x2,y-y2,z-z2])-xShift)
                   atomVectors.append(np.array([x-x2,y-y2,z-z2])-yShift)
                   atomVectors.append(np.array([x-x2,y-y2,z-z2])-xShift-yShift)
                   atomVectors.append(np.array([x-x2,y-y2,z-z2])+xShift)
                   atomVectors.append(np.array([x-x2,y-y2,z-z2])+yShift)
                   atomVectors.append(np.array([x-x2,y-y2,z-z2])+xShift+yShift)
                   atomVectors.append(np.array([x-x2,y-y2,z-z2])-2*xShift)
                   atomVectors.append(np.array([x-x2,y-y2,z-z2])-2*yShift)
                   atomVectors.append(np.array([x-x2,y-y2,z-z2])-2*xShift-
2*yShift)
                   atomVectors.append(np.array([x-x2,y-y2,z-z2])+2*xShift)
                   atomVectors.append(np.array([x-x2,y-y2,z-z2])+2*yShift)
                   atomVectors.append(np.array([x-x2,y-y2,z-
z2])+2*xShift+2*yShift)
                   if index != oldIndex:
                      for atomVector in atomVectors:
                         dotted = np.dot(atomVector/np.linalg.norm(atomVector),
normal/np.linalg.norm(normal))
                         if dotted>=.98:
                            addIt = False
                            keep.remove(atom)
                            keep.append([x,y,z,index])
                            if z not in maxZs:
                               maxZs[z]=1
                            else:
                               maxZs[z]+=1
                            if maxZs[z2]==1:
                               del maxZs[z2]
                            else:
                               maxZs[z2]-=1
                            kept.append(index)
```

```
                        kept.remove(oldIndex)
                        breakAgain=True
                        break
                    if dotted<0:
                        addIt=False

                if breakAgain==True:
                    break
            if addIt == True and index not in kept:
                keep.append([x,y,z,index])
                kept.append(index)
                if z not in maxZs:
                    maxZs[z]=1
                else:
                    maxZs[z]+=1
            elif index not in kept:
                keep.append([x,y,z,index])
                kept.append(index)
                if z not in maxZs:
                    maxZs[z]=1
                else:
                    maxZs[z]+=1
            index+=1
        loop+=1
    areaCounter = 0
    defaultHeight = max(maxZs)-min(maxZs)
    areaCounter = len(maxZs)
    defaultArea = strt.volume/strt.lattice.c/areaCounter*3
    elements = Composition(strt.formula).elements
    strtR = copy.deepcopy(strt)
    i=0
    for x in strtR.frac_coords:
        strtR.translate_sites(i,[0,0,2*(.5-x[2])])
        i+=1
    # Returns: the slab structure as an object, a list of surface sites,
    # a list of all sites, a list of elements in the slab, the minimum thickness,
    # the maximum thickness, and the ligand object. Also returned are the
suggested
    # default area for threeAtomAdsorb, default thickness from top atom of
surface layer,
    # the vector representing the normal, and the slab inverted (so the bottom
surface is
    # now the top) as a structure object
```

```
        return [strt, kept,sites, elements,min_thick,min_vac,ligand],
defaultArea,defaultHeight, normal, strtR

    def
oneAtomAdsorb(params,adatom_on_lig,spin,displacement,rotation,angleToRot):
        # First loop for generating adsorbates above atoms
        # Attach the ligands to the slab.
        # param: The set of parameters returned by setUp
        # adatom_on_lig: The adatom of the ligand that you want attaching
        #               to the surface
        # displacement: The distance between the adsorbate and the surface atom
        # rotation: The direction to rotate the adsorbate, defined by the a,b,c
        #          lattice vectors.
        # angleToRot: The angle to rotate the adsorbate along the rotation direction

        strt,keep,sites,elements,min_thick,min_vac,ligand = params
        structsToContinue = []
        for site in keep:
            iface = Interface(strt, hkl=[0,0,1],
                              min_thick=min_thick, min_vac=min_vac,
start_from_slab=True,
                              ligand=ligand, displacement=displacement, scell_nmax =
10,

adatom_on_lig=adatom_on_lig,adsorb_on_species=str(elements[0]),
                              primitive=False,from_ase=True,rot=spin)
            iface.set_top_atoms()
            topAtoms = iface.top_atoms
            failed=True
            if topAtoms ==[]:
               while topAtoms == [] and failed==True:
                  for ele in elements:
                     iface = Interface(strt, hkl=[0,0,1],
                                       min_thick=min_thick, min_vac=min_vac,
start_from_slab=True,
                                       ligand=ligand, displacement=displacement, scell_nmax =
10,

adatom_on_lig=adatom_on_lig,adsorb_on_species=str(ele),
                                       primitive=False, from_ase=True,rot=spin)
                     iface.to('poscar','TEMP.vasp')
                     iface.set_top_atoms()
                     if iface.top_atoms != []:
```

```
                    failed=False
                    break
                if iface.top_atoms == []:
                    failed = True

            iface.cover_surface2([site], rotation,angleToRot)
            iface.sort()
            # extract bare slab
            iface_slab = iface.slab
            iface_slab.sort()
            # set selective dynamics flags as required
            true_site = [1, 1, 1]
            false_site = [0, 0, 0]
            sd_flag_iface = []
            sd_flag_slab = []
            # selective dynamics flags for the interface
            for i in iface.sites:
                sd_flag_iface.append(false_site)
            # selective dynamics flags for the bare slab
            for i in iface_slab.sites:
                sd_flag_slab.append(false_site)
            interface_poscar = Poscar(iface, selective_dynamics=sd_flag_iface)
            slab_poscar = Poscar(iface_slab, selective_dynamics=sd_flag_slab)
            # poscars without selective dynamics flag
            iface.to('poscar', 'POSCAR_interface_'+str(site+1)+'.vasp')
            iface_slab.to('poscar', 'POSCAR_slab.vasp')

interface_poscar.write_file("POSCAR_interface_with_sd_"+str(site+1)+".vasp")
            slab_poscar.write_file("POSCAR_slab_with_sd.vasp")
            structsToContinue.append(iface)
 #      structsToContinue =
StructureMatcher().group_structures(structsToContinue,anonymous=False)
        return structsToContinue
    def twoAtomAdsorb(param,length, adatom_on_lig, spin, displacement,
heightDiff,rotation,angleToRot):
        # Second loop for generating adsorbates between atoms
        # param: The set of parameters returned by setUp
        # length: the maximum distance between the two atoms considered for
adsorption.
        #         In units of Angstroms
        # adatom_on_lig: The adatom of the ligand that you want attaching
        #                to the surface
        # displacement: The distance between the adsorbate and the surface atom
```

```
    # heightDiff: The maximum difference in heighet between two atoms
considered for
    #           adsorption. In unites of Angstroms
    # rotation: The direction to rotate the adsorbate, defined by the a,b,c
    #          lattice vectors.
    # angleToRot: The angle to rotate the adsorbate along the rotation direction
    strt,keep,sites,elements,min_thick,min_vac,ligand = params
    complete = []
    structsToContinue = []
    for site1 in keep:
       for site2 in keep:
          if site1!=site2:
             complete.append([site1,site2])
             # create a minimum distance adsorbate must be from the other atoms
             radCount=0
             radius=0
             for ele in elements:
                radius+=Element(ele).atomic_radius/2
                radCount+=1
             radius_criteria = radius/radCount
             if abs(sites[site1][2]-sites[site2][2])!=0:
                translation = sites[site2]-sites[site1]
                diffTrans = copy.deepcopy(translation)
                if diffTrans[0]==0:
                   diffTrans=diffTrans[1:]
                elif diffTrans[1]==0:
                   diffTrans=[diffTrans[0]]
                   diffTrans.append(translation[2])
                elif diffTrans[2]==0:
                   diffTrans=diffTrans[:2]
                if np.linalg.norm(diffTrans)<=length and
translation[2]<=heightDiff:
                   newPosition = sites[site1]+translation/2
                   skip = False
                   for site in keep:
                      if np.linalg.norm(translation)<radius_criteria:
                         skip = True
                         break
                   if skip != True:
                      iface = Interface(strt, hkl=[0,0,1],
                                  min_thick=min_thick, min_vac=min_vac,
start_from_slab=True,
```

```
                                    ligand=ligand,
displacement=displacement+translation[2]/2, scell_nmax = 10,

adatom_on_lig=adatom_on_lig,adsorb_on_species=str(elements[0]),
                                    x_shift=translation[0]/2, y_shift=translation[1]/2,
                                    primitive=False,from_ase=True,rot=spin)
                        iface.set_top_atoms()
                        topAtoms = iface.top_atoms
                        failed=False
                        if topAtoms ==[]:
                           while topAtoms == [] and failed==False:
                              for ele in elements:
                                 iface = Interface(strt, hkl=[0,0,1],
                                          min_thick=min_thick, min_vac=min_vac,
start_from_slab=True,
                                          ligand=ligand,
displacement=displacement+translation[2]/2, scell_nmax = 10,

adatom_on_lig=adatom_on_lig,adsorb_on_species=str(elements[0]),
                                          x_shift=translation[0]/2,
y_shift=translation[1]/2,
                                          primitive=False,from_ase=True,rot=spin)
                                 iface.to('poscar','TEMP.vasp')
                                 iface.set_top_atoms()
                                 if iface.top_atoms != []:
                                    break
                              if iface.top_atoms == []:
                                 failed = True
                        iface.cover_surface2([site1],rotation,angleToRot)
                        iface.sort()
                        iface_slab = iface.slab
                        iface_slab.sort()
                        true_site = [1, 1, 1]
                        false_site = [0, 0, 0]
                        sd_flag_iface = []
                        sd_flag_slab = []
                        # selective dynamics flags for the interface
                        for i in iface.sites:
                           sd_flag_iface.append(false_site)
                        # selective dynamics flags for the bare slab
                        for i in iface_slab.sites:
                           sd_flag_slab.append(false_site)
```

```
                        interface_poscar = Poscar(iface,
selective_dynamics=sd_flag_iface)
                        slab_poscar = Poscar(iface_slab,
selective_dynamics=sd_flag_slab)
                        # poscars without selective dynamics flag
                    #   iface.to('poscar', 'POSCAR_interface_'+str(x)+'.vasp')
                        iface.to('poscar', 'POSCAR_interface_trans'+str(site1+1)+'-
'+str(site2+1)+'.vasp')
                        iface_slab.to('poscar', 'POSCAR_slab.vasp')
                        # poscars with selective dynamics flag

interface_poscar.write_file("POSCAR_interface_with_sd_"+str(site1+1)+'-
'+str(site2+1)+".vasp")
                        slab_poscar.write_file("POSCAR_slab_with_sd.vasp")
                        structsToContinue.append(iface)
 #      structsToContinue =
StructureMatcher().group_structures(structsToContinue,anonymous=False)
       return structsToContinue

   def threeAtomAdsorb(params,area, adatom_on_lig,spin, displacement,
heightDiff,angleRestricts,rotation,angleToRot):
       # Third loop for generating adsorbates between atoms
       # param: The set of parameters returned by setUp
       # area: The maximum area formed by the three atoms considered for
adsorption.
       #        In units of Angstroms^2
       # adatom_on_lig: The adatom of the ligand that you want attaching
       #              to the surface
       # displacement: The distance between the adsorbate and the surface atom
       # heightDiff: The maximum difference in heighet between two atoms
considered for
       #            adsorption. In units of Angstroms.
       # angleRestricts: [minimum, maximum] angles between the three atoms
considered
       #               for adsorption
       # rotation: The direction to rotate the adsorbate, defined by the a,b,c
       #          lattice vectors.
       # angleToRot: The angle to rotate the adsorbate along the rotation direction
       strt,keep,sites,elements,min_thick,min_vac,ligand = params
       minAngle, maxAngle = angleRestricts
       structsToContinue = []
       radius = 0
       radCount=0
```

```
for ele in elements:
    radius+=Element(ele).atomic_radius
    radCount+=1
radius_criteria = radius/radCount
for site1 in keep:
    for site2 in keep:
        for site3 in keep:
            if site1!=site2 and site2!=site3 and site1!=site3:
                x1,y1,z1=sites[site1]
                x2,y2,z2=sites[site2]
                x3,y3,z3=sites[site3]
                if abs(z1-z2)<heightDiff and \
                  abs(z1-z3)<heightDiff and \
                  abs(z2-z3)<heightDiff:
                   newPosition= (sites[site1]+sites[site2]+sites[site3])/3
                   skip = False
                   a = np.array(sites[site2]-sites[site1])
                   b = np.array(sites[site3]-sites[site1])
                   if np.linalg.norm(np.cross(a,b))>area:
                      skip = True
                   if skip==False:
                      ang1 = strt.get_angle(site1,site2,site3)
                      ang2 = strt.get_angle(site2, site3, site1)
                      ang3 = strt.get_angle(site3, site1, site2)
                      if ang1 > 180:
                         ang1-=180
                      if ang2 > 180:
                         ang2-=180
                      if ang3 > 180:
                         ang3-=180
                      angles = [ang1, ang2, ang3]
                      for angle in angles:
                         if angle<minAngle or angle>maxAngle:
                            skip=True
                            break
                   translation = newPosition-sites[site1]
                   for site in keep:
                      if np.linalg.norm(translation)<radius_criteria:
                         skip = True
                         break
                   if skip != True:
                      iface = Interface(strt, hkl=[0,0,1],
```

```
                                    min_thick=min_thick, min_vac=min_vac,
start_from_slab=True,
                                    ligand=ligand,
displacement=displacement+translation[2], scell_nmax = 10,

adatom_on_lig=adatom_on_lig,adsorb_on_species=str(elements[0]),
                                    x_shift=translation[0], y_shift=translation[1],
                                    primitive=False, from_ase=True,rot=spin)
                        iface.set_top_atoms()
                        topAtoms = iface.top_atoms
                        failed=False
                        if topAtoms ==[]:
                            while topAtoms == [] and failed==False:
                                for ele in elements:
                                    iface = Interface(strt, hkl=[0,0,1],
                                                min_thick=min_thick, min_vac=min_vac,
start_from_slab=True,
                                                ligand=ligand,
displacement=displacement+translation[2], scell_nmax = 10,

adatom_on_lig=adatom_on_lig,adsorb_on_species=str(elements[0]),
                                                x_shift=translation[0],
y_shift=translation[1],
                                                primitive=False, from_ase=True,rot=spin)
                                    iface.to('poscar','TEMP.vasp')
                                    iface.set_top_atoms()
                                    if iface.top_atoms != []:
                                        break
                                if iface.top_atoms == []:
                                    failed = True
                        iface.cover_surface2([site1],rotation,angleToRot)
                        iface.sort()
                        iface_slab = iface.slab
                        iface_slab.sort()
                        true_site = [1, 1, 1]
                        false_site = [0, 0, 0]
                        sd_flag_iface = []
                        sd_flag_slab = []
                        # selective dynamics flags for the interface
                        for i in iface.sites:
                            sd_flag_iface.append(false_site)
                        # selective dynamics flags for the bare slab
                        for i in iface_slab.sites:
```

```
                        sd_flag_slab.append(false_site)
                    interface_poscar = Poscar(iface,
selective_dynamics=sd_flag_iface)
                    slab_poscar = Poscar(iface_slab,
selective_dynamics=sd_flag_slab)
                    # poscars without selective dynamics flag
                    iface.to('poscar', 'POSCAR_interface_trans'+str(site1+1)+'-
'+str(site2+1)+'-'+str(site3+1)+'.vasp')
                    iface_slab.to('poscar', 'POSCAR_slab.vasp')
                    # poscars with selective dynamics flag

interface_poscar.write_file("POSCAR_interface_with_sd_"+str(site1+1)+'-
'+str(site2+1)+'-'+str(site3+1)+".vasp")
                    slab_poscar.write_file("POSCAR_slab_with_sd.vasp")
                    structsToContinue.append(iface)
#       structsToContinue =
StructureMatcher().group_structures(structsToContinue,anonymous=False)
    return structsToContinue

  def writeStruct(structList, hkl_name, ligandName):
    # will generate directories and write POSCAR files to them
    # in the directory the function is run in. Takes a list of
    # structure objects as input.
    structsListSingle = []
    if len(structList)==1:
      structsListSingle=structList
    elif len(structList)==0:
      print('EMPTY LIST. CANNOT RUN')
      exit()
    else:
      for sublist in structList:
        for struct in sublist:
          structsListSingle.append(struct)
    finalStructs =
StructureMatcher().group_structures(structsListSingle,anonymous=False)
    i = 1
    os.mkdir(hkl_name+'_'+ligandName)
    os.chdir(hkl_name+'_'+ligandName)
    for struct in finalStructsList:
      os.mkdir(str(i))
      struct[0].to('poscar',str(i)+'/POSCAR')
      i+=1
    os.chdir('../')
```

```
def orgDataConvergence():
    # Organize the data from a slabConvergence. Run in
    # the directory where slabConvergence was run. The
    # data will be in the directory where this function was
    # run and will be called "resultsC.txt"
    comps = []
    thicks = []
    spaces = []
    diction = {}
    for comp in os.listdir('.'):
        if os.path.isdir(comp) and 'bulk' not in comp:
            comps.append(comp)

    bulkAtom = 0
    for x in open('bulk/POSCAR','r').readlines()[6].split():
        bulkAtom+=int(x)
    bulkE = float(open('bulk/OSZICAR','r').readlines()[-1].split()[4])/bulkAtom
    with open('resultsC.txt','w') as f:
        f.write('{0:^10}'.format('Thickness')+' | {0:^8}'.format('vacPad')+' |
{0:^10}'.format('eV/atom')+' \n')
    for comp in comps:
        energy = 0
        try:
            print(comp)
            thick = comp[:comp.index('_')]
            space = comp[comp.index('_')+1:]
            if open(comp+'/OSZICAR','r').readlines()[-1].split()[0].isdigit():
                energy = open(comp+'/OSZICAR','r').readlines()[-1].split()[4]
            if energy != 0:
                numAtoms = 0
                posLines = open(comp+'/POSCAR','r').readlines()
                nums = posLines[6].split()
                a = np.array(posLines[2].split())
                b = np.array(posLines[3].split())
                for num in nums:
                    numAtoms+=int(num)
                if thick not in diction:
                    diction[thick] = {}
                diction[thick][space]=((float(energy)-bulkE*numAtoms))
        #       diction[thick][space]=((float(energy)-
bulkE*numAtoms))/np.cross(a,b)#/8.785
                if thick not in thicks:
                    thicks.append(thick)
```

```
            if space not in spaces:
                spaces.append(space)
        except:
            pass
    thicks = sorted(thicks, key=float)
    spaces = sorted(spaces,key=float)
    for thick in thicks:
        for space in spaces:
            if space in diction[thick]:
                with open('resultsC.txt','a') as f:
                    f.write('{0:10}'.format(thick)+' | {0:8}'.format(space)+' |
{0:<10}'.format(diction[thick][space])+' \n')

def orgDataSlabs():
    # Will organize data generated by makeGA_slabs or makeMP_slabs.
    # Run in the directory where the slabs are located. The data will
    # be organized in a file called "resultsS.txt".

    with open('resultsS.txt','w') as f:
        f.write('{0:^10}'.format('Index')+' | {0:^10}'.format('eV/atom')+' \n')
    for comp in os.listdir('.'):
        if os.path.isdir(comp):
            if os.path.exists(comp+'/bulk/OSZICAR') and
open(comp+'/bulk/OSZICAR','r').readlines()[-1].split()[0].isdigit():
                os.chdir(comp)
                hkls = []
                with open('../resultsS.txt','a') as f:
                    f.write('-------'+comp+'------- \n')
                diction = {}
                for hkl in os.listdir('.'):
                    if os.path.isdir(hkl) and 'bulk' not in hkl:
                        hkls.append(hkl)

                bulkAtom = 0
                for x in open('bulk/POSCAR','r').readlines()[6].split():
                    bulkAtom+=int(x)
                bulkE = float(open('bulk/OSZICAR','r').readlines()[-
1].split()[4])/bulkAtom
                hklData = {}
                for hkl in hkls:
                    energy = 0
                    try:
                        print(comp)
```

```
                    oLines = open(hkl+'/OSZICAR','r').readlines()
                    if oLines[-1].split()[0].isdigit():
                        energy = oLines[-1].split()[4]
                    if energy != 0:
                        numAtoms = 0
                        posLines = open(hkl+'/POSCAR','r').readlines()
                        nums = posLines[6].split()
                        a = [float(x) for x in np.array(posLines[2].split())]
                        b = [float(x) for x in np.array(posLines[3].split())]
                        for num in nums:
                            numAtoms+=int(num)
                        hklData[hkl]=((float(energy)-
bulkE*numAtoms)/np.linalg.norm(np.cross(a,b))/2)

                except:
                    pass
            for hkl in hklData:
                with open('../resultsS.txt','a') as f:
                    f.write('{0:10}'.format(hkl)+' | {0:<10}'.format(hklData[hkl])+' \n')
            with open('../resultsS.txt','a') as f:
                f.write('\n')
            os.chdir('../')

def orgDataAdsorb():
    # Will organize adsorbate data generated by .
    # Run in the directory where the slabs are located. The data will
    # be organized in a file called "resultsA.txt".
    if os.path.exists('adsorbate/OSZICAR') and
open('adsorbate/OSZICAR','r').readlines()[-1].split()[0].isdigit():
        adsorbateEnergy = float(open('adsorbate/OSZICAR','r').readlines()[-
1].split()[4])
    else:
        print('NO ADSORBATE PRESENT. ENDING FUNCTION.')
        exit()
    with open('resultsA.txt','w') as f:
        f.write('{0:^10}'.format('Index')+' | {0:^10}'.format('eV/atom')+' \n')
    for comp in os.listdir('.'):
        if os.path.isdir(comp):

            if os.path.exists(comp+'/bulk/OSZICAR') and
open(comp+'/bulk/OSZICAR','r').readlines()[-1].split()[0].isdigit():
                os.chdir(comp)
                hkls = []
```

```
            with open('../resultsA.txt','a') as f:
               f.write('-------'+comp+'------- \n')
            diction = {}
            for hkl in os.listdir('.'):
               if os.path.isdir(hkl) and 'bulk' not in hkl:
                  hkls.append(hkl)

            bulkAtom = 0
            for x in open('bulk/POSCAR','r').readlines()[6].split():
               bulkAtom+=int(x)
            bulkE = float(open('bulk/OSZICAR','r').readlines()[-
1].split()[4])/bulkAtom
            hklData = {}
            adsorbData = {}
            for hkl in hkls:
               energy = 0
               try:
                  print(comp)
                  oLines = open(hkl+'/OSZICAR','r').readlines()
                  if oLines[-1].split()[0].isdigit():
                     energy = float(oLines[-1].split()[4])
                     hklData[hkl]=energy
               except:
                  pass
               try:
                  adsorbData[hkl] = {}
                  for site in os.listdir(hkl):
                     if os.path.isdir(hkl+'/'+site):
                        print(1)
                        energy=0
                        aLines = open(hkl+'/'+site+'/OSZICAR','r').readlines()
                        print(2)
                        if aLines[-1].split()[0].isdigit():
                           energy = float(aLines[-1].split()[4])
                        print(3,energy)
                        if energy!=0:
                           print(4)
                           adsorbData[hkl][site] = energy-adsorbateEnergy-
hklData[hkl]
                           print(5)
               except:
                  pass
            print(adsorbData)
```

```
            for hkl in adsorbData:
                with open('../resultsA.txt','a') as f:
                    f.write('{0:10}'.format(hkl)+' \n')
                    for site in adsorbData[hkl]:
                        f.write('{0:10}'.format(site)+' |
{0:10}'.format(str(round(adsorbData[hkl][site],5)))+' \n')
            with open('../resultsA.txt','a') as f:
                f.write('\n')
            os.chdir('../')
```

Appendix F. MPInterface Modification

This appendix appears in its original form, without editorial change.

This appendix is comprised of an addition of “cover_surface2” to MPInterface’s mpinterfaces.interface *Interface* class. This addition enables the rotation of adsorbates on the slab surface.

```
def cover_surface2(self, site_indices, rotation,
angleToRot):
        """
        puts the ligand molecule on the given list of
site indices
        """
        num_atoms = len(self.ligand)
        normal = self.normal
        # get a vector that points from one atom in
the botton plane
        # to one atom on the top plane. This is
required to make sure
        # that the surface normal points outwards from
the surface on
        #  to which we want to adsorb the ligand
        vec_vac = self.cart_coords[self.top_atoms[0]]
- \
            self.cart_coords[self.bottom_atoms[0]]
        # mov_vec = the vector along which the ligand
will be displaced
        mov_vec = normal * self.displacement
        angle = get_angle(vec_vac, self.normal)
        # flip the orientation of normal if it is not
pointing in
        # the right direction.
        if (angle > 90):
            normal_frac =
self.lattice.get_fractional_coords(normal)
```

```
            normal_frac[2] = -normal_frac[2]
            normal =
self.lattice.get_cartesian_coords(normal_frac)
            mov_vec = normal * self.displacement
        # get the index corresponding to the given
atomic species in
        # the ligand that will bond with the surface
on which the
        # ligand will be adsorbed
        adatom_index =
self.get_index(self.adatom_on_lig)
        adsorbed_ligands_coords = []
        # set the ligand coordinates for each
adsorption site on
        # the surface
        for sindex in site_indices:
            # align the ligand wrt the site on the
surface to which
            # it will be adsorbed
            origin = self.cart_coords[sindex]

self.ligand.translate_sites(list(range(num_atoms)),
                                        origin -
self.ligand[

adatom_index].coords)
            # displace the ligand by the given amount
in the direction
            # normal to surface
```

```
self.ligand.translate_sites(list(range(num_atoms)),
mov_vec)

            # vector pointing from the adatom_on_lig
to the

            # ligand center of mass

            vec_adatom_cm = self.ligand.center_of_mass
- \

                self.ligand[adatom_index].coords

            # rotate the ligand with respect to a
vector that is

            # normal to the vec_adatom_cm and the
normal to the surface

            # so that the ligand center of mass is
aligned along the

            # outward normal to the surface

            origin = self.ligand[adatom_index].coords

            angle = get_angle(vec_adatom_cm, normal)

            if 1 < abs(angle % 180) < 179:

                # For angles which are not 0 or 180,

                # perform a rotation about the origin
along an axis

                # perpendicular to both bonds to align
bonds.

                axis = np.cross(vec_adatom_cm, normal)

                op =
SymmOp.from_origin_axis_angle(origin, axis, angle)

                self.ligand.apply_operation(op)

            elif abs(abs(angle) - 180) < 1:

                # We have a 180 degree angle.
```

```
                # Simply do an inversion about the
origin
                for i in range(len(self.ligand)):
                    self.ligand[i] =
(self.ligand[i].species_and_occu,
                                        origin - (

self.ligand[i].coords - origin))
            axis = np.cross(vec_adatom_cm, normal)
            op =
SymmOp.from_origin_axis_angle(origin,rotation,angleToR
ot)
            self.ligand.apply_operation(op)
            # x - y - shifts
            x = self.x_shift
            y = self.y_shift
            rot = self.rot
            op =
SymmOp.from_origin_axis_angle(self.cart_coords[sindex]
,[1,0,0],rot[0])
            self.ligand.apply_operation(op)
            op =
SymmOp.from_origin_axis_angle(self.cart_coords[sindex]
,[0,1,0],rot[1])
            self.ligand.apply_operation(op)
            op =
SymmOp.from_origin_axis_angle(self.cart_coords[sindex]
,[0,0,1],rot[2])
            self.ligand.apply_operation(op)
            if x:
```

```
self.ligand.translate_sites(list(range(num_atoms)),

np.array([x, 0, 0]))
            if y:

self.ligand.translate_sites(list(range(num_atoms)),

np.array([0, y, 0]))
# 3d numpy array

adsorbed_ligands_coords.append(self.ligand.cart_coords
)
            # extend the slab structure with the
adsorbant atoms
        adsorbed_ligands_coords =
np.array(adsorbed_ligands_coords)
        for j in range(len(site_indices)):

[self.append(self.ligand.species_and_occu[i],
adsorbed_ligands_coords[j, i, :],
                         coords_are_cartesian=True)
             for i in range(num_atoms)]
```

Appendix G. Python Script for reciprocal.py to Generate KPOINT Files

This appendix appears in its original form, without editorial change.

```
import os

def makeKPOINTS(twoD, MeshType,Length,
desired_directory):

    #Input Variables

    MeshType = MeshType

    TwoDimensional = twoD

    l = Length

    input_POSCAR = desired_directory+'POSCAR'

    output_KPOINTS = desired_directory+'KPOINTS'


    POSCAR = open(input_POSCAR, 'r')

    lines = POSCAR.readlines()

    scale =
Decimal.from_float(float(lines[1].split()[0]))


    #Define original lattice vectors


    a1 = lines[2].split()

    a2 = lines[3].split()

    a3 = lines[4].split()


    a11 = scale*Decimal.from_float(float(a1[0]))

    a12 = scale*Decimal.from_float(float(a1[1]))

    a13 = scale*Decimal.from_float(float(a1[2]))


    a21 = scale*Decimal.from_float(float(a2[0]))

    a22 = scale*Decimal.from_float(float(a2[1]))

    a23 = scale*Decimal.from_float(float(a2[2]))
```

```
    a31 = scale*Decimal.from_float(float(a3[0]))
    a32 = scale*Decimal.from_float(float(a3[1]))
    a33 = scale*Decimal.from_float(float(a3[2]))

    #Calculate the determinant

    det = a11*(a22*a33-a23*a32)-a12*(a21*a33-
a23*a31)+a13*(a21*a32-a22*a31)

    #Calculate reciprocal vectors

    b11 = (a22*a33-a23*a32)/det
    b12 = (a21*a33-a23*a31)/det
    b13 = (a21*a32-a22*a31)/det

    b21 = (a12*a33-a13*a32)/det
    b22 = (a11*a33-a13*a31)/det
    b23 = (a11*a32-a12*a31)/det

    b31 = (a12*a23-a13*a22)/det
    b32 = (a11*a23-a13*a21)/det
    b33 = (a11*a22-a12*a21)/det

    with open(output_KPOINTS, 'w') as file:
        kpoints_x = ''
        kpoints_y = ''
```

```
        kpoints_z = ''
        file.write('Automatic mesh\n')
        file.write( '0\n')
        kpoints_x =
round(float(l*Decimal.sqrt(b11*b11+b12*b12+b13*b13)))
        kpoints_y =
round(float(l*Decimal.sqrt(b21*b21+b22*b22+b23*b23)))
        kpoints_z =
round(float(l*Decimal.sqrt(b31*b31+b32*b32+b33*b33)))
        if TwoDimensional == True:
            kpoints_z =1
        file.write(MeshType + '\n')
        file.write(str(int(kpoints_x)) + " " +
str(int(kpoints_y)) + " " + str(int(kpoints_z)))
        print kpoints_x, kpoints_y, kpoints_z
if __name__ == '__main__':
            makeKPOINTS(True, 'Gamma',50,'')
```

List of Symbols, Abbreviations, and Acronyms

API	application program interface
ARL	US Army Research Laboratory
DFT	density functional theory
DOD	Department of Defense
GA	genetic algorithm
GASP	genetic algorithm for structure prediction
HPCMP	High Performance Computing Modernization Program
pymatgen	python materials genomics
VASP	Vienna ab initio Simulation Package

1 (PDF) DEFENSE TECHNICAL INFORMATION CTR DTIC OCA

2 (PDF) DIR ARL
IMAL HRA
RECORDS MGMT
RDRL DCL
TECH LIB

1 (PDF) GOVT PRINTG OFC
A MALHOTRA

1 (PDF) UNIV OF FLORIDA
J PAUL

2 (PDF) UNIV OF ILLINOIS URBANA-CHAMPAIGN
S CHAUDHURI
P PRIYA

4 (PDF) ARL
RDRL WMM B
M TSCHOPP
E HERNANDEZ
RDRL WMM F
K LIMMER
S COLEMAN